htmap Documentation

Release 0.6.1

CHTC

Jan 08, 2024

GETTING STARTED

HTMap is a library that wraps the process of mapping Python function calls out to an HTCondor pool. It provides tools for submitting, managing, and processing the output of arbitrary functions.

Our goal is to provide as transparent an interface as possible to high-throughput computing resources so that you can spend more time thinking about your own code, and less about how to get it running on a cluster.

Running a map over a Python function is as easy as

```
import htmap


def double(x):
    return 2 * x


doubled = list(htmap.map(double, range(10)))
print(doubled)
# [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

If you're just getting started, jump into the first tutorial: *First Steps*.

Happy mapping!

Installation
: Installing HTMap

Note: Bug reports and feature requests should go on our GitHub issue tracker.

Tutorials
: Tutorials on using HTMap.

Dependency Management
: Information about how to manage your what your code depends on (e.g., other Python packages).

API Reference
: Public API documentation.

CLI Reference
: Use of the HTMap CLI.

Using HTCondor with HTMap
: Tips on using HTMap with HTCondor

Tips and Tricks
: Useful tips & tricks on the API.

FAQ
: These questions are asked, sometimes frequently.

Settings
: Documentation for the various settings.

Version History
: New features, bug fixes, and known issues by version.

Contributing and Developing
: How to contribute to HTMap, how to set up a development environment, how HTMap works under the hood, etc.

CHAPTER

ONE

INSTALLATION

- On Unix/Linux systems, running `pip install htmap` from the command line should suffice.
- On Windows, there's an added dependency of HTCondor (to get access to the HTCondor Python bindings). After that, use the `pip install --no-deps`.

The introductory tutorials can be run on Binder, requiring no setup on your part.

Basic usage only requires installation of HTMap "submit-side". Anything more advanced like checkpointing or output file transfers will require installation on the execute nodes. For more information and to ensure your code will run correctly execute-side see *Dependency Management*.

You may need to append `--user` to the `pip` command if you do not have permission to install packages directly into the Python installation you are using. Recent versions of `pip` will do this automatically when necessary.

CHAPTER

TWO

TUTORIALS

Attention: The most convenient way to go through these tutorials is through Binder, which requires no setup on your part:

First Steps
: If this is your first time using HTMap, start here!

Basic Mapping
: An introduction to the basics of HTMap.

Working with Files
: Sending additional files with your maps.

Map Options
: How to tell the pool what to do with your map.

Advanced Mapping
: More (and better) ways to create maps.

Error Handling
: What do when something goes wrong.

2.1 First Steps

2.1.1 Setup

The fastest and easiest way to make sure you have a working setup (as described below) is to go through these tutorials on Binder

The second-easiest way is to run the tutorials in a Docker container on your computer. Run

```
docker run -p 8888:8888 htcondor/htmap-tutorials
```

and follow the instructions it gives you to get into the Jupyter environment. Then go to `tutorials/first-steps.ipynb` in the file browser and open it to get back to this point.

Alternatively, you might want to immediately start running HTMap on your HTCondor pool. This tutorial assumes that you've already installed HTMap on your HTCondor pool's submit node, or have access to HTMap through a JupyterHub server connected to an HTCondor pool or similar. See *How do I install HTMap?* for details!

This tutorial also assumes that you're working in a Jupyter Notebook. It will work just as well in the Python REPL. Later, once you get a hang things, you'll be ready to use HTMap in scripts as well. Either way, you'll need to be on a computer that can submit jobs to an HTCondor pool.

This tutorial assumes that you have already set up your dependency management, as described in *Dependency Management*. If your HTCondor pool supports Docker, you'll be good to go with the default settings.

The tutorials in this series are written inside Juypter Notebooks. If you click the "View page source" link in the upper right corner, you'll be able to grab the raw `.ipynb` file yourself and step through it along with the tutorial.

2.1.2 The Problem

Suppose you've been given the task of writing a function that doubles numbers, like this:

[1]:
```
def double(x):
    return 2 * x
```

If you want to double a list of numbers, you might do something like

[2]:
```
doubled = [double(x) for x in range(10)]
print(doubled)
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

or we can use the built-in function `map()`, which applies a function to each element of an iterable (like a list):

[3]:
```
mapped = map(double, range(10))
print(mapped)
doubled = list(mapped)
print(doubled)
```

```
<map object at 0x7f7ae8393390>
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

In both cases, `doubled` is the list `[0, 2, 4, ...]`. The reason we need the `list` call is that `map` actually returns an *iterator* over the results, not the results themselves. So you need to iterate over it to get the output, which is what `list` does: iterate over its input and put the elements in a list.

Now suppose that, for some reason, you want to double *a lot* of numbers. So many numbers that you can't bear to do all the work on your own computer. It takes days to multiply all the numbers, and if your program crashes halfway through, you lose all of of your progress and have to start over. You're losing sleep, and your boss is breathing down your neck because they need those numbers doubled *now*.

Luckily, you remember that you have access to an HTCondor high-throughput computing pool. Since each of your function calls is isolated from all the others, the computers in the pool don't need to talk to each other at

all, and you can achieve a huge speedup. The pool can run your code on hundreds or thousands of computers simultaneously, storing the inputs and outputs for you and recovering from individual errors gracefully. It's the perfect solution.

The problem is: *how do you get your code running in the pool?*

2.1.3 The Solution

With HTMap, it's like this:

[4]:
```
import htmap

mapped = htmap.map(double, range(10))
print(mapped)
doubled = list(mapped)
print(doubled)
```

```
Created map super-busy-dog with 10 components
Map(tag = super-busy-dog)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

It may take some time for the second `print` to run. During that time, the individual **components** of your **map** are being run out on the cluster on execute nodes. Once they all finish, you'll get the list of numbers back. As you can see, the output is identical to what you would get from running the function locally.

In the *next tutorial* we'll start digging into the extra features that HTMap provides on top of this basic functionality.

2.2 Basic Mapping

2.2.1 Tags

In the previous tutorial, we used HTMap like this:

[1]:
```
import htmap

def double(x):
    return 2 * x
```

[2]:
```
mapped = htmap.map(double, range(10))
print(mapped)
doubled = list(mapped)
print(doubled)
```

```
Created map dark-puny-robe with 10 components
Map(tag = dark-puny-robe)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

In particular, we used the htmap.map function to create our map. This function creates an object that behaves a lot like the iterator returned by the built-in map function. To get our output, we iterated over it using `list`.

You may have noticed that the map has a **tag** associated with it. HTMap generated this tag for us because we didn't provide one, and because we didn't provide one, marked the map as **transient**, as opposed to **persistent**. Transient maps are for quick tests where we don't care too much about organization. Persistent maps are for longer-running maps where we want to keep our work organized by giving things real names. If you don't plan on using your map for more than one session, you can probably get away with a transient map. If you're going to step away from the computer and come back, we recommend giving it a real tag.

The map we created above is transient:

[3]:
```
print(mapped.is_transient)
```

```
True
```

To create a persistent map, we need to give our map our map a tag:

[4]:
```
another_map = htmap.map(double, range(10), tag = 'dbl')
print(another_map)
print(another_map.is_transient)
```

```
Created map dbl with 10 components
Map(tag = dbl)
False
```

We can also "retag" a map to give it a new tag. If you tag a transient map, it becomes persistent.

[5]:
```
mapped.retag('a-new-tag')
print(mapped)
print(mapped.is_transient)
```

```
Map(tag = a-new-tag)
False
```

2.2.2 Working with Maps

The object that was returned by htmap.map is a htmap.Map. It gives us a window into the map as it is running, and lets us use the output once the map is finished.

For example, we can print the status of the map:

[6]:
```
stringified = htmap.map(str, range(10), tag = 'str')
print(stringified.status())
```

```
Created map str with 10 components
Map str (10 components): HELD = 0 | ERRORED = 0 | IDLE = 10 | RUNNING = 0 |␣
↪COMPLETED = 0
```

We can wait for the map to finish:

[7]:
```
stringified.wait(show_progress_bar = True)
```

```
str: 100%|##########| 10/10 [00:09<00:00,  1.11component/s]
```

There are many ways to iterate over maps:

[8]:
```
print(list(stringified))

for d in stringified:
    print(d)
```

```
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
0
1
2
3
4
5
6
7
8
9
```

If we ever lose our reference to it, we can grab a new reference to it using htmap.load, giving it the tag of the map we want:

[9]:
```
new_ref = htmap.load('str')

print(new_ref)
print(new_ref == stringified)
print(new_ref is stringified)  # maps are singletons
```

```
Map(tag = str)
True
True
```

Maps can be recovered from an entirely different Python interpreter session as well. Suppose you close Python and go on vacation. You come back and you want to look at your map again, but you've forgotten what you called it. Just ask HTMap for a list of your tags:

[10]:
```
print(htmap.get_tags())
```

```
('dbl', 'str', 'a-new-tag')
```

Ok, well, technically it was a tuple, but we'll have to live with it.

HTMap can also print a pretty table showing the status of your maps:

[11]:
```
htmap.map(str, range(5))  # new transient map
print(htmap.status())
```

```
Created map breezy-happy-hand with 5 components
Tag                  HELD  ERRORED  IDLE  RUNNING  COMPLETED  Local Data  Max␣
↪Memory  Max Runtime  Total Runtime
a-new-tag               0        0     0        0         10     63.9 KB     41.0␣
↪MB      0:00:00        0:00:00
dbl                     0        0     0        0         10     63.9 KB     41.0␣
↪MB      0:00:00        0:00:00
str                     0        0     0        0         10     63.5 KB     41.0␣
↪MB      0:00:00        0:00:00
* breezy-happy-hand     0        0     5        0          0     19.8 KB      0.0␣
↪B       0:00:00        0:00:00
```

Note that transient maps have a * in front of their tags.

The status message tells us about how many components of our map are in each of the five most common component states:

- **Idle** - component is waiting to run
- **Running** - component is currently executing remotely
- **Completed** - component is finished executing and output is available
- **Held** - HTCondor has noticed a problem with the component and is not letting it run
- **Errored** - there was an error in your code, and HTMap has brought back error information

The status of each component of your map is available using the map attribute `component_statuses`:

[12]:
```
print(new_ref.component_statuses)
```

```
[<ComponentStatus.COMPLETED: 'COMPLETED'>, <ComponentStatus.COMPLETED: 'COMPLETED
↪'>, <ComponentStatus.COMPLETED: 'COMPLETED'>, <ComponentStatus.COMPLETED:
↪'COMPLETED'>, <ComponentStatus.COMPLETED: 'COMPLETED'>, <ComponentStatus.
↪COMPLETED: 'COMPLETED'>, <ComponentStatus.COMPLETED: 'COMPLETED'>,
↪<ComponentStatus.COMPLETED: 'COMPLETED'>, <ComponentStatus.COMPLETED:
↪'COMPLETED'>, <ComponentStatus.COMPLETED: 'COMPLETED'>]
```

We'll discuss what to do about held and errored components and how to interact with component statuses in the *Error Handling* tutorial.

Tags are *unique*: if we try to create another map with a tag we've already used, it will fail:

[13]:
```
new_map = htmap.map(double, range(10), tag = 'dbl')
```

```
---------------------------------------------------------------------------
TagAlreadyExists                          Traceback (most recent call last)
```

(continues on next page)

(continued from previous page)

```
<ipython-input-13-397c48e54a47> in <module>
----> 1 new_map = htmap.map(double, range(10), tag = 'dbl')

~/htmap/htmap/mapping.py in map(func, args, map_options, tag)
     86         func,
     87         args_and_kwargs,
---> 88         map_options = map_options,
     89     )
     90

~/htmap/htmap/mapping.py in create_map(tag, func, args_and_kwargs, map_options)
    276
    277     tags.raise_if_tag_is_invalid(tag)
--> 278     tags.raise_if_tag_already_exists(tag)
    279
    280     logger.debug(f'Creating map {tag} ...')

~/htmap/htmap/tags.py in raise_if_tag_already_exists(tag)
     59     """Raise a :class:`htmap.exceptions.TagAlreadyExists` if the ``tag``␣
↪already exists."""
     60     if tag_file_path(tag).exists():
---> 61         raise exceptions.TagAlreadyExists(f'The requested tag "{tag}"␣
↪already exists. Load the Map with htmap.load("{tag}"), or remove it using␣
↪htmap.remove("{tag}").')
     62
     63

TagAlreadyExists: The requested tag "dbl" already exists. Load the Map with␣
↪htmap.load("dbl"), or remove it using htmap.remove("dbl").
```

As the error message indicates, if we want to re-use the tag `dbl`, we need to remove the old map first:

[14]:
```
old_map = htmap.load('dbl')
old_map.remove()
```

htmap.Map.remove deletes all traces of the map. **It can never be recovered**. Be careful when using it!

The module-level shortcut htmap.remove lets you skip the intermediate step of getting the actual Map, if you don't already have it.

Now we can re-use the map ID:

[15]:
```
new_map = htmap.map(double, range(10), tag = 'dbl')
new_map.wait(show_progress_bar = True)
print(list(new_map))
```

```
dbl:   0%|          | 0/10 [00:00<?, ?component/s]
```

```
Created map dbl with 10 components
dbl: 100%|##########| 10/10 [00:07<00:00,  1.42component/s]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

2.2.3 Map Builders

So far we've been avoiding any functions that needed to be mapped over keyword arguments, or that had more than one positional argument. `htmap.map` is not really the ideal tool for working with functions that have more than one argument, and it does not support varying more than one argument at all.

A much more ergonomic way to build up a complex map is a **map builder**. A map builder lets you build a map via individual function calls. Call htmap.build_map as a context manager to get the builder, then call the builder as if it were the mapped function itself:

[16]:
```
def power(base, exponent):
    return base ** exponent

with htmap.build_map(power) as pow_builder:
    for base in range(1, 5):  # bases are 1, 2, 3, 4
        for exponent in range(1, 4):  # exponents are 1, 2, 3
            pow_builder(base, exponent)

powered = pow_builder.map
print(list(powered))  # 1^1, 1^2, 1^3, 2^1, 2^2, 2^3, 3^1 ...
```

```
Created map harsh-happy-ring with 12 components
[1, 1, 1, 2, 4, 8, 3, 9, 27, 4, 16, 64]
```

The map builder catches the function calls and turns them into a map. The map is created when the `with` block ends, at which point you can grab the actual htmap.Map from the builder's `map` attribute.

In the *next tutorial*, we'll see how to tell HTMap to bring a local file along to the execute node.

2.3 Working with Files

High-throughput computing often involves analyzing data stored in files. For many simple cases, HTMap can automatically work with files that you specify as arguments of your function without (much) special treatment.

Let's start with "Hello world!" example:

[1]:
```
from pathlib import Path

def read_file(path: Path):
    return path.read_text()
```

This function takes in a pathlib.Path, reads it, and returns its contents. Let's make a file and see how it works:

[2]:
```
hi_path = Path.cwd() / 'hi.txt'
print(hi_path)
hi_path.write_text('Hello world!')
```

```
/home/jovyan/tutorials/hi.txt
```

[2]:
```
12
```

[3]:
```
print(read_file(hi_path))
```

```
Hello world!
```

(pathlib has a steeper learning curve than `os.path`, but it's well worth the effort!)

Now, let's start mapping. In this case, the map call is barely different than the original function call, but we need to set up the inputs correctly. The trick is that, instead of a `pathlib.Path`, we need to use a htmap.TransferPath. `htmap.TransferPath` is a drop-in replacement for `pathlib.Path` in every way, except for HTMap's special treatment of it.

HTMap will detect that we used an `htmap.TransferPath` in a map as long as it is an argument or keyword argument of the function, or stored in a primitive container (`list`, `dict`, `set`, `tuples`) and automatically transfer the named file to wherever the function executes.

[4]:
```
import htmap

bye_path = htmap.TransferPath.cwd() / 'bye.txt'
bye_path.write_text('Have a nice day!')
```

```
16
```

[5]:
```
map = htmap.map(read_file, [bye_path])
print(map.get(0))  # map.get will wait until the result is ready
```

```
Created map puny-thin-echo with 1 components
Have a nice day!
```

2.3.1 Multiple Files

To see how we can transfer a container full of files, let's write a simple clone of the unix `cat` program, which concatenates files. It takes a single argument which is a list of files to be concatenated, and returns the concatenated files as a string.

[6]:
```
def cat(files):
    file_contents = (file.read_text() for file in files)
    return ''.join(file_contents)
```

Let's write some test files…

[7]:
```
cwd = htmap.TransferPath.cwd()
paths = [
    cwd / 'start.txt',
    cwd / 'middle.txt',
    cwd / 'end.txt',
]
parts = [
    'The quick brown ',
    'fox jumps over ',
    'the lazy dog!',
]
for path, part in zip(paths, parts):
    path.write_text(part)
```

… and run a map!

[8]:
```
m = htmap.map(cat, [paths])  # this creates a single map component with the list␣
↪of paths as the argument
print(m.get(0))
```

```
Created map red-bland-tub with 1 components
The quick brown fox jumps over the lazy dog!
```

If the "output" of your map function needs to be a file instead of a Python object (or you produce files that you need back submit-side for whatever reason), you'll want to look at the Output Files recipe once you're done with the tutorials.

In the *next tutorial* we'll learn how to tell HTCondor about what resources our map components require, as well as another HTCondor configuration they need.

2.4 Map Options

2.4.1 Requesting Resources

The most common kind of map option you'll probably need to work with are the ones for requests resources. HTMap makes fairly conservative default choices about the resources required by your map components. If your function needs a lot of resources, such as memory or disk space, you will need to communicate this to HTMap.

Suppose we need to transfer a huge input file that we need to read into memory, so we need a lot of memory and disk space available on the execute node. We'll request 200 MB of RAM, 10 GB of disk space, and send our input file.

[1]:
```
from pathlib import Path
import htmap

def read_huge_file(file):
    contents = Path(file).read_text()

    # do stuff

    return contents  # we'll just return the contents here, but imagine this is␣
↪the result of processing
```

[2]:
```
huge_file = htmap.TransferPath.cwd() / 'huge_file.txt'
huge_file.write_text('only a few words, but use your imagination')
```

[2]:
```
42
```

(Don't panic! `write_text()` returns the number of bytes written.)

And here's our map call:

[3]:
```
processed = htmap.map(
    read_huge_file,
    [huge_file],
    map_options = htmap.MapOptions(
        request_memory = '100MB',
        request_disk = '1GB',
    ),
)

print(processed.get(0))
```

```
Created map breezy-thick-beak with 1 components
only a few words, but use your imagination
```

`request_memory` and `request_disk` were passed as single strings. Since they are single strings, they will be treated as **fixed options** and applied to every component. The other kind of option is **variadic**, which lets you specify some option for each component of the map individually. For exampe, if we wanted a different amount of RAM for each component, we could pass a list of strings to `request_memory`, one for each component:

[4]:
```
multiple = htmap.map(
    read_huge_file,
    [huge_file, huge_file, huge_file],
    map_options = htmap.MapOptions(
        request_memory = ['10MB', '20MB', '30MB'],
        request_disk = '1GB',
    ),
)
print(list(multiple))
```

```
Created map tall-soft-stream with 3 components
['only a few words, but use your imagination', 'only a few words, but use your␣
↪imagination', 'only a few words, but use your imagination']
```

2.4.2 The Kitchen Sink

HTMap also supports **arbitrary HTCondor submit descriptors**, like you would see in a submit file. Just pass them as keyword arguments to a htmap.MapOptions, keeping in mind that you can use standard ClassAd interpolation and that the same fixed/variadic behavior applies.

If that didn't make sense, don't worry about it! The whole point of HTMap is to avoid needing to know too much about submit descriptors.

The *next tutorial* discusses more convenient and flexible way of defining your maps.

2.5 Advanced Mapping

So far we've built our maps using the top-level mapping functions. These functions are useful for tutorials, but don't give us the full flexibility that we might need when working with arbitrary Python functions. They're also sometimes inconvenient to use, especially if you don't like typing the names of your functions over and over. The tools described in this tutorial fix those problems.

2.5.1 Starmap

Back in *Working With Files*, we noted that htmap.map was only able to handle functions that took a single argument. To work with a function that took two arguments, we needed to use htmap.build_map to build up the map inside a loop.

Sometimes, you don't want to loop. *htmap.starmap* provides the flexibility to completely specify the positional and keyword arguments for every component without needing an explicit `for`-loop.

Unfortunately, that looks like this:

[1]:
```
import htmap

def power(x, p = 1):
        return x ** p
```

[2]:
```
starmap = htmap.starmap(
    func = power,
    args = [
        (1,),
        (2,),
        (3,),
    ],
    kwargs = [
        {'p': 1},
        {'p': 2},
        {'p': 3},
    ],
)

print(list(starmap))  # [1, 4, 27]
```

```
Created map proper-short-stream with 3 components
[1, 4, 27]
```

A slightly more pleasant but less obvious way to construct the arguments would be like this:

[3]:
```
starmap = htmap.starmap(
    func = power,
    args = ((x,) for x in range(1, 4)),
    kwargs = ({'p': p} for p in range(1, 4)),
)

print(list(starmap))  # [1, 4, 27]
```

```
Created map light-soggy-idea with 3 components
[1, 4, 27]
```

But that isn't really a huge improvement. Sometimes you'll need the power and compactness of `starmap`, but we recommend htmap.build_map for general use.

2.5.2 Mapped Functions

If you're tired of typing `htmap.map` all the time, create a htmap.MappedFunction using the htmap.mapped decorator:

[4]:
```
@htmap.mapped
def double(x):
    return 2 * x

print(double)
```

```
MappedFunction(func = <function double at 0x7f750c0653b0>, map_options = {})
```

The resulting `MappedFunction` has methods that correspond to all the mapping functions, but with the function already filled in.

For example:

[5]:
```
doubled = double.map(range(10))

print(list(doubled))  # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
Created map coy-burst-area with 10 components
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

The real utility of mapped functions is that they can carry default map options, which are **inherited** by any maps created from them. For example, if we know that a certain function will always need a large amount of memory and disk space, we can specify it for **any** map like this:

[6]:
```
@htmap.mapped(
    map_options = htmap.MapOptions(
        request_memory = '200MB',
        request_disk = '1GB',
    )
)
def big_list(_):
    big = list(range(1_000_000))  # imagine this is way bigger...
    return big
```

Now our `request_memory` and `request_disk` will be set for each map, without needing to specify it in the `MapOptions` of each individual `map` call. We can still override the setting for a certain map by manually passing htmap.MapOptions.

See htmap.MapOptions for some notes about how these inherited map options behave.

2.5.3 Non-Primitive Function Arguments

So far we've mostly limited our mapped function arguments to Python primitives like integers or strings. However, HTMap will work with almost any Python object. For example, we can use a custom class as a function argument. Maybe we have some data on penguins, and we want to write a `Penguin` class to encapsulate that data:

[7]:
```
class Penguin:
    def __init__(self, name, height, weight):
        self.name = name
        self.height = height
        self.weight = weight

    def analyze(self):
        return f'{self.name} is {self.height} inches tall and weighs {self.
↪weight} pounds'

    def eat(self):
        print('mmm, yummy fish')

    def fly(self):
        raise TypeError("penguins can't fly!")
```

[8]:
```
penguins = [
    Penguin('Gwendolin', height = 73, weight = 51),
    Penguin('Gweniffer', height = 59, weight = 43),
    Penguin('Gary', height = 64, weight = 52),
]
```

[9]:
```
map = htmap.map(
    lambda p: p.analyze(),  # an anonmyous function; see https://docs.python.org/
↪3/tutorial/controlflow.html#lambda-expressions
    penguins,
    tag = 'penguin-stats',
)

map.wait(show_progress_bar = True)
```

```
penguin-stats:   0%|          | 0/3 [00:00<?, ?component/s]
Created map penguin-stats with 3 components
penguin-stats: 100%|##########| 3/3 [00:03<00:00,  1.00s/component]
```

[10]:
```
for stats in map:
    print(stats)
```

```
Gwendolin is 73 inches tall and weighs 51 pounds
Gweniffer is 59 inches tall and weighs 43 pounds
```

(continues on next page)

(continued from previous page)

```
Gary is 64 inches tall and weighs 52 pounds
```

Specialized data structures like `numpy` arrays and `pandas` dataframes can also be used as function arguments. When in doubt, just try it!

In the *next tutorial* we'll finally address the most important part of programming: what to do when things go wrong!

2.6 Error Handling

2.6.1 Holds

In previous tutorials we mentioned that HTMap is able to track the status of your components and inform you about something called a "hold". A hold occurs when HTCondor notices something wrong about your map component. Perhaps an input file is missing, or your component tried to use a file that didn't exist.

The last one is easy to force, so let's do it and see what happens:

[1]:
```
import htmap

@htmap.mapped
def foo(_):  # _ is a perfectly legal argument name, often used to mean "I don't␣
↪actually use it"
    return "I didn't get held!"
```

[2]:
```
path = htmap.TransferPath('this-file-does-not-exist.txt')
will_get_held = foo.map(
    [path],
)
```

```
Created map angry-husky-law with 1 components
```

We know that the component will fail, but HTMap won't know about it until we try to look at the output:

[3]:
```
print(will_get_held.get(0))
```

```
---------------------------------------------------------------------------
MapComponentHeld                          Traceback (most recent call last)
<ipython-input-3-68dfbf32680e> in <module>
----> 1 print(will_get_held.get(0))

~/htmap/htmap/maps.py in _protect(self, *args, **kwargs)
```

(continues on next page)

(continued from previous page)

```
     43         if not self.exists:
     44             raise exceptions.MapWasRemoved(f'Cannot call {method} for␣
↪map {self.tag} because it has been removed')
---> 45         return method(self, *args, **kwargs)
     46
     47     return _protect

~/htmap/htmap/maps.py in get(self, component, timeout)
    390             If ``None``, wait forever.
    391         """
--> 392         return self._load_output(component, timeout = timeout)
    393
    394     def __getitem__(self, item: int) -> Any:

~/htmap/htmap/maps.py in _load_output(self, component, timeout)
    341             raise IndexError(f'Tried to get output for component
↪{component}, but map {self.tag} only has {len(self)} components')
    342
--> 343         self._wait_for_component(component, timeout)
    344
    345         status_and_result = htio.load_objects(self._output_file_
↪path(component))

~/htmap/htmap/maps.py in _wait_for_component(self, component, timeout)
    307                 break
    308             elif component_status is state.ComponentStatus.HELD:
--> 309                 raise exceptions.MapComponentHeld(f'Component {component}
↪ of map {self.tag} is held: {self.holds[component]}')
    310
    311             if timeout is not None and (time.time() >= start_time +␣
↪timeout):

MapComponentHeld: Component 0 of map angry-husky-law is held: [13] Error from␣
↪slot1_6@1bea834c10a5: SHADOW at 172.17.0.2 failed to send file(s) to <172.17.0.
↪2:33571>: error reading from /home/jovyan/tutorials/this-file-does-not-exist.
↪txt: (errno 2) No such file or directory; STARTER failed to receive file(s)␣
↪from <172.17.0.2:9618>
```

Yikes! HTMap has raised an exception to inform us that a component of our map got held. It also tells us why HTCondor held the component: `error reading from /home/jovyan/tutorials/this-file-does-not-exist: (errno 2) No such file or directory; STARTER failed to receive file(s) from <172.17.0.2:9618>`.

This time around the hold reason is pretty clear: a local file that HTCondor expected to exist didn't. We could fix the problem by creating the file, and then releasing the map, which tells HTCondor to try again:

[4]:
```
path.touch()  # this creates an empty file
```

Now the map will run successfully. We tell HTMap to "release" the hold, allowing the map to continue running.

[5]:
```
will_get_held.release()
print(will_get_held.get(0))
```

```
I didn't get held!
```

Debugging holds

Unfortunately, holds will often not be so easy to resolve. Sometimes they are simply ephemeral errors that can be resolved by releasing the map without changing anything. But sometimes you'll need to talk to your HTCondor pool administrator to figure out what's going wrong.

Sometimes these errors are caused by additional parameters specified in your `~/.htmaprc` file. Are you sure `~/.htmaprc` has the intended parameters?

If you're feeling really adventurous, look at files in the directory `~/.htmap/`. The standard output and error files are contained within this directory. This might help solve your problem.

2.6.2 Execution Errors

HTMap can also detect Python exceptions that occur during component execution. To see this in action, let's define a function where a component will have a problem:

[6]:
```
@htmap.mapped
def inverse(x):
    return 1 / x
```

When `x = 0`, `inverse(x)` will fail with a `ZeroDivisionError`. If we run it locally, the error will halt execution and drop a traceback into our laps:

[7]:
```
inverse(0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-7-7538d73c586c> in <module>
----> 1 inverse(0)

~/htmap/htmap/mapped.py in __call__(self, *args, **kwargs)
     50     def __call__(self, *args, **kwargs):
     51         """Call the function as normal, locally."""
---> 52         return self.func(*args, **kwargs)
     53
     54     def map(
```

(continues on next page)

(continued from previous page)

```
<ipython-input-6-769ac4dfb4b6> in inverse(x)
      1 @htmap.mapped
      2 def inverse(x):
----> 3     return 1 / x

ZeroDivisionError: division by zero
```

The traceback has a lot of critically-useful information in it. In fact, it tells us exactly the line that raised the error (remember that tracebacks should be read in reverse - the last block of source code is where the error began).

HTMap is able to transport this kind of information back from an executing component, but like the regular output of a map we won't see it until we try to load up the output for the failed component. We'll make a one-component map to demonstrate what happens:

[8]:
```
bad_map = inverse.map([0])
bad_map.get(0)
```

```
Created map fair-sly-drone with 1 components
```

```
---------------------------------------------------------------------------
MapComponentError                         Traceback (most recent call last)
<ipython-input-8-d23b8117e4db> in <module>
      1 bad_map = inverse.map([0])
----> 2 bad_map.get(0)

~/htmap/htmap/maps.py in _protect(self, *args, **kwargs)
     43         if not self.exists:
     44             raise exceptions.MapWasRemoved(f'Cannot call {method} for␣
↪map {self.tag} because it has been removed')
---> 45         return method(self, *args, **kwargs)
     46
     47     return _protect

~/htmap/htmap/maps.py in get(self, component, timeout)
    390             If ``None``, wait forever.
    391         """
--> 392         return self._load_output(component, timeout = timeout)
    393
    394     def __getitem__(self, item: int) -> Any:

~/htmap/htmap/maps.py in _load_output(self, component, timeout)
    348             return next(status_and_result)
    349         elif status == 'ERR':
--> 350             raise exceptions.MapComponentError(f'Component {component}␣
↪of map {self.tag} encountered error while executing. Error report:\n{self._
```

(continues on next page)

(continued from previous page)

```
↪load_error(component).report()}')
    351         else:
    352             raise exceptions.InvalidOutputStatus(f'Output status {status}
↪ is not valid')

MapComponentError: Component 0 of map fair-sly-drone encountered error while␣
↪executing. Error report:
==========  Start error report for component 0 of map fair-sly-drone  ==========
Landed on execute node 1bea834c10a5 (172.17.0.2) at 2020-05-21 17:45:40.954824

Python executable is /opt/conda/bin/python3 (version 3.7.6)
with installed packages
  alembic==1.4.2
  async-generator==1.10
  attrs==19.3.0
  backcall==0.1.0
  bleach==3.1.4
  blinker==1.4
  brotlipy==0.7.0
  certifi==2020.4.5.1
  certipy==0.1.3
  cffi==1.14.0
  chardet==3.0.4
  click==7.1.2
  click-didyoumean==0.0.3
  cloudpickle==1.4.1
  colorama==0.4.3
  conda==4.8.2
  conda-package-handling==1.6.0
  cryptography==2.9.2
  cursor==1.3.4
  decorator==4.4.2
  defusedxml==0.6.0
  entrypoints==0.3
  halo==0.0.29
  htchirp==1.0
  htcondor==8.9.6
  -e git+https://github.com/htcondor/htmap.
↪git@e0fd6de94fcad0295ae674e5479fac51cf57f34f#egg=htmap
  idna==2.9
  importlib-metadata==1.6.0
  ipykernel==5.2.1
  ipython @ file:///home/conda/feedstock_root/build_artifacts/ipython_
↪1588362967322/work
  ipython-genutils==0.2.0
  jedi==0.17.0
```

(continues on next page)

(continued from previous page)

```
Jinja2==2.11.2
json5==0.9.0
jsonschema==3.2.0
jupyter-client==6.1.3
jupyter-core==4.6.3
jupyter-telemetry==0.0.5
jupyterhub==1.1.0
jupyterlab==2.1.1
jupyterlab-server==1.1.1
log-symbols==0.0.14
Mako==1.1.0
MarkupSafe==1.1.1
mistune==0.8.4
nbconvert==5.6.1
nbformat==5.0.6
nbstripout==0.3.7
notebook==6.0.3
oauthlib==3.0.1
pamela==1.0.0
pandocfilters==1.4.2
parso==0.7.0
pexpect==4.8.0
pickleshare==0.7.5
prometheus-client==0.7.1
prompt-toolkit==3.0.5
ptyprocess==0.6.0
pycosat==0.6.3
pycparser==2.20
pycurl==7.43.0.5
Pygments==2.6.1
PyJWT==1.7.1
pyOpenSSL==19.1.0
pyrsistent==0.16.0
PySocks==1.7.1
python-dateutil==2.8.1
python-editor==1.0.4
python-json-logger==0.1.11
pyzmq==19.0.0
requests==2.23.0
ruamel-yaml==0.15.80
ruamel.yaml.clib==0.2.0
Send2Trash==1.5.0
six==1.14.0
spinners==0.0.24
SQLAlchemy==1.3.16
termcolor==1.1.0
```

(continues on next page)

(continued from previous page)

```
  terminado==0.8.3
  testpath==0.4.4
  toml==0.10.0
  tornado==6.0.4
  tqdm==4.46.0
  traitlets==4.3.3
  urllib3==1.25.9
  wcwidth==0.1.9
  webencodings==0.5.1
  zipp==3.1.0

Scratch directory contents are
  /home/jovyan/.condor/local/execute/dir_461/.chirp.config
  /home/jovyan/.condor/local/execute/dir_461/_htmap_user_transfer
  /home/jovyan/.condor/local/execute/dir_461/.job.ad
  /home/jovyan/.condor/local/execute/dir_461/_condor_stderr
  /home/jovyan/.condor/local/execute/dir_461/.machine.ad
  /home/jovyan/.condor/local/execute/dir_461/func
  /home/jovyan/.condor/local/execute/dir_461/_condor_stdout
  /home/jovyan/.condor/local/execute/dir_461/0.in
  /home/jovyan/.condor/local/execute/dir_461/_htmap_transfer
  /home/jovyan/.condor/local/execute/dir_461/_htmap_do_output_transfer
  /home/jovyan/.condor/local/execute/dir_461/_htmap_transfer_plugin_cache
  /home/jovyan/.condor/local/execute/dir_461/condor_exec.exe
  /home/jovyan/.condor/local/execute/dir_461/.update.ad

Exception and traceback (most recent call last):
  File "<ipython-input-6-769ac4dfb4b6>", line 3, in inverse
    return 1 / x

    Local variables:
      x = 0

  ZeroDivisionError: division by zero

===========  End error report for component 0 of map fair-sly-drone  ===========
```

Neat! This traceback is, unfortunately, harder to read than the other one. We need to ignore everything above `MapComponentError: component 0 of map <tag> encountered error while executing. Error report:` - it's just about the internal error that HTMap is raising to propagate the error to us. The real error is the stuff below `========= Start error report for component 0 of map <tag> =========`.

Since we're trying to debug remotely, HTMap has gathered some metadata about the HTCondor "execute node" where the component was running. First it tell us where it is and when the component started executing. Next, the report tells us about the Python environment that was used to execute your function, including a list of installed packages. We also get a listing of the contents of the working directory - in this example,

because we didn't add any extra input files, it's just a bunch of files that HTCondor and HTMap are using.

The meat of the error is the last thing in the error report. We get roughly the same information that we got in the local traceback, but we also get a printout of the local variables in each stack frame.

Since the local HTMap error is raised as soon as it finds a bad component, you may find it convenient to look at *all* of the error reports for your map (hopefully not too many!). htmap.Map.error_reports provides exactly this functionality:

[9]:
```
worse_map = inverse.map([0, 0, 0])
worse_map.wait(errors_ok = True)  # wait for all of the components to hit the␣
↪error
for report in worse_map.error_reports():
    print(report + '\n')
```

```
Created map firm-vast-oven with 3 components
==========  Start error report for component 0 of map firm-vast-oven  ==========
Landed on execute node 1bea834c10a5 (172.17.0.2) at 2020-05-21 17:45:44.454503

Python executable is /opt/conda/bin/python3 (version 3.7.6)
with installed packages
  alembic==1.4.2
  async-generator==1.10
  attrs==19.3.0
  backcall==0.1.0
  bleach==3.1.4
  blinker==1.4
  brotlipy==0.7.0
  certifi==2020.4.5.1
  certipy==0.1.3
  cffi==1.14.0
  chardet==3.0.4
  click==7.1.2
  click-didyoumean==0.0.3
  cloudpickle==1.4.1
  colorama==0.4.3
  conda==4.8.2
  conda-package-handling==1.6.0
  cryptography==2.9.2
  cursor==1.3.4
  decorator==4.4.2
  defusedxml==0.6.0
  entrypoints==0.3
  halo==0.0.29
  htchirp==1.0
  htcondor==8.9.6
  -e git+https://github.com/htcondor/htmap.
↪git@e0fd6de94fcad0295ae674e5479fac51cf57f34f#egg=htmap
  idna==2.9
```

(continues on next page)

(continued from previous page)

```
  importlib-metadata==1.6.0
  ipykernel==5.2.1
  ipython @ file:///home/conda/feedstock_root/build_artifacts/ipython_
↪1588362967322/work
  ipython-genutils==0.2.0
  jedi==0.17.0
  Jinja2==2.11.2
  json5==0.9.0
  jsonschema==3.2.0
  jupyter-client==6.1.3
  jupyter-core==4.6.3
  jupyter-telemetry==0.0.5
  jupyterhub==1.1.0
  jupyterlab==2.1.1
  jupyterlab-server==1.1.1
  log-symbols==0.0.14
  Mako==1.1.0
  MarkupSafe==1.1.1
  mistune==0.8.4
  nbconvert==5.6.1
  nbformat==5.0.6
  nbstripout==0.3.7
  notebook==6.0.3
  oauthlib==3.0.1
  pamela==1.0.0
  pandocfilters==1.4.2
  parso==0.7.0
  pexpect==4.8.0
  pickleshare==0.7.5
  prometheus-client==0.7.1
  prompt-toolkit==3.0.5
  ptyprocess==0.6.0
  pycosat==0.6.3
  pycparser==2.20
  pycurl==7.43.0.5
  Pygments==2.6.1
  PyJWT==1.7.1
  pyOpenSSL==19.1.0
  pyrsistent==0.16.0
  PySocks==1.7.1
  python-dateutil==2.8.1
  python-editor==1.0.4
  python-json-logger==0.1.11
  pyzmq==19.0.0
  requests==2.23.0
  ruamel-yaml==0.15.80
```

(continues on next page)

(continued from previous page)

```
  ruamel.yaml.clib==0.2.0
  Send2Trash==1.5.0
  six==1.14.0
  spinners==0.0.24
  SQLAlchemy==1.3.16
  termcolor==1.1.0
  terminado==0.8.3
  testpath==0.4.4
  toml==0.10.0
  tornado==6.0.4
  tqdm==4.46.0
  traitlets==4.3.3
  urllib3==1.25.9
  wcwidth==0.1.9
  webencodings==0.5.1
  zipp==3.1.0

Scratch directory contents are
  /home/jovyan/.condor/local/execute/dir_492/.chirp.config
  /home/jovyan/.condor/local/execute/dir_492/_htmap_user_transfer
  /home/jovyan/.condor/local/execute/dir_492/.job.ad
  /home/jovyan/.condor/local/execute/dir_492/_condor_stderr
  /home/jovyan/.condor/local/execute/dir_492/.machine.ad
  /home/jovyan/.condor/local/execute/dir_492/func
  /home/jovyan/.condor/local/execute/dir_492/_condor_stdout
  /home/jovyan/.condor/local/execute/dir_492/0.in
  /home/jovyan/.condor/local/execute/dir_492/_htmap_transfer
  /home/jovyan/.condor/local/execute/dir_492/_htmap_do_output_transfer
  /home/jovyan/.condor/local/execute/dir_492/_htmap_transfer_plugin_cache
  /home/jovyan/.condor/local/execute/dir_492/condor_exec.exe
  /home/jovyan/.condor/local/execute/dir_492/.update.ad

Exception and traceback (most recent call last):
  File "<ipython-input-6-769ac4dfb4b6>", line 3, in inverse
    return 1 / x

    Local variables:
      x = 0

  ZeroDivisionError: division by zero

===========  End error report for component 0 of map firm-vast-oven  ===========

==========  Start error report for component 1 of map firm-vast-oven  ==========
Landed on execute node 1bea834c10a5 (172.17.0.2) at 2020-05-21 17:45:44.216714
```

(continues on next page)

(continued from previous page)

```
Python executable is /opt/conda/bin/python3 (version 3.7.6)
with installed packages
  alembic==1.4.2
  async-generator==1.10
  attrs==19.3.0
  backcall==0.1.0
  bleach==3.1.4
  blinker==1.4
  brotlipy==0.7.0
  certifi==2020.4.5.1
  certipy==0.1.3
  cffi==1.14.0
  chardet==3.0.4
  click==7.1.2
  click-didyoumean==0.0.3
  cloudpickle==1.4.1
  colorama==0.4.3
  conda==4.8.2
  conda-package-handling==1.6.0
  cryptography==2.9.2
  cursor==1.3.4
  decorator==4.4.2
  defusedxml==0.6.0
  entrypoints==0.3
  halo==0.0.29
  htchirp==1.0
  htcondor==8.9.6
  -e git+https://github.com/htcondor/htmap.
↪git@e0fd6de94fcad0295ae674e5479fac51cf57f34f#egg=htmap
  idna==2.9
  importlib-metadata==1.6.0
  ipykernel==5.2.1
  ipython @ file:///home/conda/feedstock_root/build_artifacts/ipython_
↪1588362967322/work
  ipython-genutils==0.2.0
  jedi==0.17.0
  Jinja2==2.11.2
  json5==0.9.0
  jsonschema==3.2.0
  jupyter-client==6.1.3
  jupyter-core==4.6.3
  jupyter-telemetry==0.0.5
  jupyterhub==1.1.0
  jupyterlab==2.1.1
  jupyterlab-server==1.1.1
  log-symbols==0.0.14
```

(continues on next page)

(continued from previous page)

```
Mako==1.1.0
MarkupSafe==1.1.1
mistune==0.8.4
nbconvert==5.6.1
nbformat==5.0.6
nbstripout==0.3.7
notebook==6.0.3
oauthlib==3.0.1
pamela==1.0.0
pandocfilters==1.4.2
parso==0.7.0
pexpect==4.8.0
pickleshare==0.7.5
prometheus-client==0.7.1
prompt-toolkit==3.0.5
ptyprocess==0.6.0
pycosat==0.6.3
pycparser==2.20
pycurl==7.43.0.5
Pygments==2.6.1
PyJWT==1.7.1
pyOpenSSL==19.1.0
pyrsistent==0.16.0
PySocks==1.7.1
python-dateutil==2.8.1
python-editor==1.0.4
python-json-logger==0.1.11
pyzmq==19.0.0
requests==2.23.0
ruamel-yaml==0.15.80
ruamel.yaml.clib==0.2.0
Send2Trash==1.5.0
six==1.14.0
spinners==0.0.24
SQLAlchemy==1.3.16
termcolor==1.1.0
terminado==0.8.3
testpath==0.4.4
toml==0.10.0
tornado==6.0.4
tqdm==4.46.0
traitlets==4.3.3
urllib3==1.25.9
wcwidth==0.1.9
webencodings==0.5.1
zipp==3.1.0
```

(continues on next page)

(continued from previous page)

```
Scratch directory contents are
  /home/jovyan/.condor/local/execute/dir_487/.chirp.config
  /home/jovyan/.condor/local/execute/dir_487/_htmap_user_transfer
  /home/jovyan/.condor/local/execute/dir_487/.job.ad
  /home/jovyan/.condor/local/execute/dir_487/_condor_stderr
  /home/jovyan/.condor/local/execute/dir_487/.machine.ad
  /home/jovyan/.condor/local/execute/dir_487/func
  /home/jovyan/.condor/local/execute/dir_487/_condor_stdout
  /home/jovyan/.condor/local/execute/dir_487/_htmap_transfer
  /home/jovyan/.condor/local/execute/dir_487/1.in
  /home/jovyan/.condor/local/execute/dir_487/_htmap_do_output_transfer
  /home/jovyan/.condor/local/execute/dir_487/_htmap_transfer_plugin_cache
  /home/jovyan/.condor/local/execute/dir_487/condor_exec.exe
  /home/jovyan/.condor/local/execute/dir_487/.update.ad

Exception and traceback (most recent call last):
  File "<ipython-input-6-769ac4dfb4b6>", line 3, in inverse
    return 1 / x

    Local variables:
      x = 0

  ZeroDivisionError: division by zero

===========  End error report for component 1 of map firm-vast-oven  ===========

==========  Start error report for component 2 of map firm-vast-oven  ==========
Landed on execute node 1bea834c10a5 (172.17.0.2) at 2020-05-21 17:45:44.383019

Python executable is /opt/conda/bin/python3 (version 3.7.6)
with installed packages
  alembic==1.4.2
  async-generator==1.10
  attrs==19.3.0
  backcall==0.1.0
  bleach==3.1.4
  blinker==1.4
  brotlipy==0.7.0
  certifi==2020.4.5.1
  certipy==0.1.3
  cffi==1.14.0
  chardet==3.0.4
  click==7.1.2
  click-didyoumean==0.0.3
  cloudpickle==1.4.1
```

(continues on next page)

(continued from previous page)

```
  colorama==0.4.3
  conda==4.8.2
  conda-package-handling==1.6.0
  cryptography==2.9.2
  cursor==1.3.4
  decorator==4.4.2
  defusedxml==0.6.0
  entrypoints==0.3
  halo==0.0.29
  htchirp==1.0
  htcondor==8.9.6
  -e git+https://github.com/htcondor/htmap.
↪git@e0fd6de94fcad0295ae674e5479fac51cf57f34f#egg=htmap
  idna==2.9
  importlib-metadata==1.6.0
  ipykernel==5.2.1
  ipython @ file:///home/conda/feedstock_root/build_artifacts/ipython_
↪1588362967322/work
  ipython-genutils==0.2.0
  jedi==0.17.0
  Jinja2==2.11.2
  json5==0.9.0
  jsonschema==3.2.0
  jupyter-client==6.1.3
  jupyter-core==4.6.3
  jupyter-telemetry==0.0.5
  jupyterhub==1.1.0
  jupyterlab==2.1.1
  jupyterlab-server==1.1.1
  log-symbols==0.0.14
  Mako==1.1.0
  MarkupSafe==1.1.1
  mistune==0.8.4
  nbconvert==5.6.1
  nbformat==5.0.6
  nbstripout==0.3.7
  notebook==6.0.3
  oauthlib==3.0.1
  pamela==1.0.0
  pandocfilters==1.4.2
  parso==0.7.0
  pexpect==4.8.0
  pickleshare==0.7.5
  prometheus-client==0.7.1
  prompt-toolkit==3.0.5
  ptyprocess==0.6.0
```

(continues on next page)

(continued from previous page)

```
  pycosat==0.6.3
  pycparser==2.20
  pycurl==7.43.0.5
  Pygments==2.6.1
  PyJWT==1.7.1
  pyOpenSSL==19.1.0
  pyrsistent==0.16.0
  PySocks==1.7.1
  python-dateutil==2.8.1
  python-editor==1.0.4
  python-json-logger==0.1.11
  pyzmq==19.0.0
  requests==2.23.0
  ruamel-yaml==0.15.80
  ruamel.yaml.clib==0.2.0
  Send2Trash==1.5.0
  six==1.14.0
  spinners==0.0.24
  SQLAlchemy==1.3.16
  termcolor==1.1.0
  terminado==0.8.3
  testpath==0.4.4
  toml==0.10.0
  tornado==6.0.4
  tqdm==4.46.0
  traitlets==4.3.3
  urllib3==1.25.9
  wcwidth==0.1.9
  webencodings==0.5.1
  zipp==3.1.0

Scratch directory contents are
  /home/jovyan/.condor/local/execute/dir_488/.chirp.config
  /home/jovyan/.condor/local/execute/dir_488/_htmap_user_transfer
  /home/jovyan/.condor/local/execute/dir_488/.job.ad
  /home/jovyan/.condor/local/execute/dir_488/_condor_stderr
  /home/jovyan/.condor/local/execute/dir_488/.machine.ad
  /home/jovyan/.condor/local/execute/dir_488/func
  /home/jovyan/.condor/local/execute/dir_488/_condor_stdout
  /home/jovyan/.condor/local/execute/dir_488/_htmap_transfer
  /home/jovyan/.condor/local/execute/dir_488/2.in
  /home/jovyan/.condor/local/execute/dir_488/_htmap_do_output_transfer
  /home/jovyan/.condor/local/execute/dir_488/_htmap_transfer_plugin_cache
  /home/jovyan/.condor/local/execute/dir_488/condor_exec.exe
  /home/jovyan/.condor/local/execute/dir_488/.update.ad
```

(continues on next page)

(continued from previous page)

```
Exception and traceback (most recent call last):
  File "<ipython-input-6-769ac4dfb4b6>", line 3, in inverse
    return 1 / x

    Local variables:
      x = 0

  ZeroDivisionError: division by zero

===========  End error report for component 2 of map firm-vast-oven  ===========
```

Unlike holds, you generally won't want to re-run components that experienced errors (they'll just fail again). Instead, an error is usually a signal that you've got a bug in your own code. Remove your map, debug the error locally, then create a new map.

2.6.3 Standard Output and Error

When handling trickier errors, you may need to look at the `stdout` and `stderr` from your map components. `stdout` and `stderr` are what you would see on the terminal if you executed your code locally - things like `print` and exceptions normally display their information there. HTMap provides access to `stdout` and `stderr` for each component through the appropriately-named attributes of your maps:

[10]:
```
import sys

@htmap.mapped
def stdx(_):
    print("Hi from stdout!")  # stdout is the default
    print("Hi from stderr!", file = sys.stderr)

m = stdx.map([None])
```

```
Created map quick-calm-stream with 1 components
```

[11]:
```
m.stdout.get(0)  # get will wait for the stdout to become available, m.stdout[0] wouldn't
```

[11]:
```
Landed on execute node 1bea834c10a5 (172.17.0.2) at 2020-05-21 17:45:47.056114 as jovyan

Scratch directory contents before run:
|- .chirp.config
|- .job.ad
|- .machine.ad
|- .update.ad
|- 0.in
```

(continues on next page)

(continued from previous page)

```
|- _condor_stderr
|- _condor_stdout
|- _htmap_do_output_transfer
|- * _htmap_transfer
|- * _htmap_transfer_plugin_cache
|- * _htmap_user_transfer
|  \- * 0
|- condor_exec.exe
\- func

Python executable is /opt/conda/bin/python3 (version 3.7.6)
with installed packages
  alembic==1.4.2
  async-generator==1.10
  attrs==19.3.0
  backcall==0.1.0
  bleach==3.1.4
  blinker==1.4
  brotlipy==0.7.0
  certifi==2020.4.5.1
  certipy==0.1.3
  cffi==1.14.0
  chardet==3.0.4
  click==7.1.2
  click-didyoumean==0.0.3
  cloudpickle==1.4.1
  colorama==0.4.3
  conda==4.8.2
  conda-package-handling==1.6.0
  cryptography==2.9.2
  cursor==1.3.4
  decorator==4.4.2
  defusedxml==0.6.0
  entrypoints==0.3
  halo==0.0.29
  htchirp==1.0
  htcondor==8.9.6
  -e git+https://github.com/htcondor/htmap.
↪git@e0fd6de94fcad0295ae674e5479fac51cf57f34f#egg=htmap
  idna==2.9
  importlib-metadata==1.6.0
  ipykernel==5.2.1
  ipython @ file:///home/conda/feedstock_root/build_artifacts/ipython_
↪1588362967322/work
  ipython-genutils==0.2.0
  jedi==0.17.0
```

(continues on next page)

(continued from previous page)

```
Jinja2==2.11.2
json5==0.9.0
jsonschema==3.2.0
jupyter-client==6.1.3
jupyter-core==4.6.3
jupyter-telemetry==0.0.5
jupyterhub==1.1.0
jupyterlab==2.1.1
jupyterlab-server==1.1.1
log-symbols==0.0.14
Mako==1.1.0
MarkupSafe==1.1.1
mistune==0.8.4
nbconvert==5.6.1
nbformat==5.0.6
nbstripout==0.3.7
notebook==6.0.3
oauthlib==3.0.1
pamela==1.0.0
pandocfilters==1.4.2
parso==0.7.0
pexpect==4.8.0
pickleshare==0.7.5
prometheus-client==0.7.1
prompt-toolkit==3.0.5
ptyprocess==0.6.0
pycosat==0.6.3
pycparser==2.20
pycurl==7.43.0.5
Pygments==2.6.1
PyJWT==1.7.1
pyOpenSSL==19.1.0
pyrsistent==0.16.0
PySocks==1.7.1
python-dateutil==2.8.1
python-editor==1.0.4
python-json-logger==0.1.11
pyzmq==19.0.0
requests==2.23.0
ruamel-yaml==0.15.80
ruamel.yaml.clib==0.2.0
Send2Trash==1.5.0
six==1.14.0
spinners==0.0.24
SQLAlchemy==1.3.16
termcolor==1.1.0
```

(continues on next page)

(continued from previous page)

```
  terminado==0.8.3
  testpath==0.4.4
  toml==0.10.0
  tornado==6.0.4
  tqdm==4.46.0
  traitlets==4.3.3
  urllib3==1.25.9
  wcwidth==0.1.9
  webencodings==0.5.1
  zipp==3.1.0

Running component 0
  <function stdx at 0x146c42004680>
with args
  (None,)
and kwargs
  {}

----- MAP COMPONENT OUTPUT START -----

Hi from stdout!

-----  MAP COMPONENT OUTPUT END  -----

Finished executing component at 2020-05-21 17:45:47.256167

Scratch directory contents after run:
|- .chirp.config
|- .job.ad
|- .machine.ad
|- .update.ad
|- 0.in
|- _condor_stderr
|- _condor_stdout
|- * _htmap_current_checkpoint
|- _htmap_do_output_transfer
|- * _htmap_transfer
|  \- 0.out
|- * _htmap_transfer_plugin_cache
|- * _htmap_user_transfer
|  \- * 0
|- condor_exec.exe
\- func
```

Note that much of the same information from the error report is included in the component `stdout` for convenience.

[12]:
```
m.stderr.get(0)
```

[12]:
```
Hi from stderr!
```

These attributes are both iterable sequences, which means that you can do something like this:

[13]:
```
@htmap.mapped
def err(x):
    print(f"Hi from stderr! {x}", file = sys.stderr)

err_map = err.map(range(5))
err_map.wait(show_progress_bar = True)

for e in err_map.stderr:
    print(e)
```

```
green-happy-year:   0%|          | 0/5 [00:00<?, ?component/s]
Created map green-happy-year with 5 components
green-happy-year: 100%|##########| 5/5 [00:04<00:00,  1.25component/s]
Hi from stderr! 0

Hi from stderr! 1

Hi from stderr! 2

Hi from stderr! 3

Hi from stderr! 4


```

[ ]:

2.7 Advanced Tutorials

Note: these tutorial can not be run with Binder

Docker Image Cookbook
: How to build HTMap-compatible Docker images.

Output Files
: How to move arbitrary files back to the submit machine, or to other locations.

Wrapping External Programs
: How to send input and output to an external (i.e., non-Python) program from inside a mapped function.

Checkpointing Maps
How to write a function that can continue from partial progress after being evicted.

Using HTMap on the Open Science Grid
How to use HTMap on the Open Science Grid.

2.7.1 Docker Image Cookbook

Docker is, essentially, a way to send a self-contained computer called a container to another person. You define the software that goes into the container, and then anyone with Docker installed on their own computer (the "host") can run your container and access the software inside without that sofware being installed on the host. This is an enormous advantage in distributed computing, where it can be difficult to ensure that software that your own software depends on ("dependencies") are installed on the computers your code actually runs on.

To use Docker, you write a **Dockerfile** which tells Docker how to generate an **image**, which is a blueprint to construct a **container**. The Dockerfile is a list of instructions, such as shell commands or instructions for Docker to copy files from the build environment into the image. You then tell Docker to "build" the image from the Dockerfile.

For use with HTMap, you then upload this image to Docker Hub, where it can then be downloaded to execute nodes in an HTCondor pool. When your HTMap component lands on an execute node, HTCondor will download your image from Docker Hub and run your code inside it using HTMap.

The following sections describe, roughly in order of increasing complexity, different ways to build Docker images for use with HTMap. Each level of complexity is introduced to solve a more advanced dependency management problem. We recommend reading them in order until reach one that works for your dependencies (each section assumes knowledge of the previous sections).

More detailed information on how Dockerfiles work can be found in the Docker documentation itself This page only covers the bare minimum to get started with HTMap and Docker.

> **Attention:** This recipe only covers using Docker for **execute-side** dependency management. You still need to install dependencies **submit-side** to launch your map in the first place!

Can I use HTMap's default image?

HTMap's default Docker image is htcondor/htmap-exec, which is itself based on`continuumio/anaconda3 <https://hub.docker.com/r/continuumio/anaconda3/>`_. It is based on Python 3 and has many useful packages pre-installed, such as `numpy`, `scipy`, and `pandas`. If your software only depends on packages included in the Anaconda distribution, you can use HTMap's default image and won't need to create your own.

I depend on Python packages that aren't in the Anaconda distribution

Attention: Before proceeding, install Docker on your computer and make an account on Docker Hub.

Let's pretend that there's a package called `foobar` that your Python function depends on, but isn't part of the Anaconda distribution. You will need to write your own Dockerfile to include this package in your Docker image.

Docker images are built in **layers**. You always start a Dockerfile by stating which existing Docker image you'd like to use as your base layer. A good choice is the same Anaconda image that HTMap uses as the default, which comes with both the `conda` package manager and the standard `pip`. Create a file named `Dockerfile` and write this into it:

```
# Dockerfile

FROM continuumio/anaconda3:latest
ENV PATH=/opt/conda/bin/:${PATH}

RUN pip install --no-cache-dir htmap

ARG USER=htmap
RUN groupadd ${USER} \
 && useradd -m -g ${USER} ${USER}
USER ${USER}
```

Each line in the Dockerfile starts with a short, capitalized word which tells Docker what kind of build instruction it is.

- `FROM` means "start with this base image".
- `RUN` means "execute these shell commands in the container".
- `ARG` means "set build argument" - it acts like an environment variable that's only set during the image build.

Lines that begin with a `#` are comments in a Dockerfile. The above lines say that we want to inherit from the image `continuumio/anaconda3:latest` and build on top of it. To be compatible with HTMap, we install `htmap` via `pip`. We also set up a non-root user to do the execution, which is important for security. Naming that user `htmap` is arbitrary and has nothing to do with the `htmap` package itself.

Now we need to tell Docker to run a shell command during the build to install `foobar` by adding one more line to the bottom of the Dockerfile.

```
# Dockerfile

FROM continuumio/anaconda3:latest
ENV PATH=/opt/conda/bin/:${PATH}
```

(continues on next page)

(continued from previous page)

```
RUN pip install --no-cache-dir htmap

ARG USER=htmap
RUN groupadd ${USER} \
 && useradd -m -g ${USER} ${USER}
USER ${USER}

# if foobar can be install via conda, use these lines
RUN conda install -y foobar \
 && conda clean -y --all

# if foobar can be installed via pip, use these lines
RUN pip install --no-cache-dir foobar
```

Some notes on the above:

- If you need to install some packages via `conda` and some via `pip`, you may need to use both types of lines.
- The `conda clean` and `--no-cache-dir` instructions for `conda` and `pip` respectively just help keep the final Docker image as small as possible.
- The `-y` options for the `conda` commands are the equivalent of answering "yes" to questions that `conda` asks on the command line, since the Docker build is non-interactive.
- A trailing `\` is a line continuation, so that first command is equivalent to running `conda install -y foobar && conda clean -y --all`, which is just `bash` shorthand for "do both of these things".

If you need install many packages, we recommend writing a `requirements.txt` file (see the Python packaging docs) and using

```
# Dockerfile

FROM continuumio/anaconda3:latest
ENV PATH=/opt/conda/bin/:${PATH}

RUN pip install --no-cache-dir htmap

ARG USER=htmap
RUN groupadd ${USER} \
 && useradd -m -g ${USER} ${USER}
USER ${USER}

COPY requirements.txt requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
```

The `COPY` build instruction tells Docker to copy the file `requirements.txt` (path relative to the build directory, explained below) to the path `requirements.txt` inside the image. Relative paths inside the container work the same way they do in the shell; the image has a "working directory" that you can set using the

`WORKDIR` instruction.

Now that we have a Dockerfile, we can tell Docker to use it to build an image. You'll need to choose a descriptive name for the image, ideally something easy to type that's related to your project (like `qubits` or `gene-analysis`). Wherever you see `<image>` below, insert that name. You'll also want to version your images by adding a "tag" after a `:`, like `<image>:v1`, `<image>:v2`, `<image>:v3`, etc. You can use any string you'd like for the tag. You'll also need to know your Docker Hub username. Wherever you see `<username>` below, insert your username, and wherever you see `<tag>`, insert your chosen version tag.

At the command line, in the directory that contains `Dockerfile`, run

```
$ docker build -t <username>/<image>:<tag> .
```

You should see the output of the build process, hopefully ending with

```
Successfully tagged <username>/<image>:<tag>
```

`<username>/<image>:<tag>` is the universal identifier for your image.

Now we need to push the image up to Docker Hub. Run

```
$ docker push <username>/<image>:<tag>
```

You'll be asked for your credentials, and then all of the data for your image will be pushed up to Docker Hub. Once this is done, you should be able to use the image with HTMap. Change your HTMap settings (see *DOCKER*) to point to your new image, and launch your maps!

I don't need most of the Anaconda distribution and want to use a lighter-weight base image

Instead of using the full Anaconda distribution, use a base Docker image that only includes the `conda` package manager:

```
# Dockerfile

FROM continuumio/miniconda3:latest
ENV PATH=/opt/conda/bin/:${PATH}

RUN pip install --no-cache-dir htmap

ARG USER=htmap
RUN groupadd ${USER} \
 && useradd -m -g ${USER} ${USER}
USER ${USER}
```

From here, install your particular dependencies as above.

If you prefer to not use `conda`, an even-barer-bones image could be produced from

```
# Dockerfile

FROM python:latest

RUN pip install --no-cache-dir htmap

ARG USER=htmap
RUN groupadd ${USER} \
 && useradd -m -g ${USER} ${USER}
USER ${USER}
```

We use `python:latest` as our base image, so we don't have `conda` anymore.

I want to use a Python package that's not on PyPI or Anaconda

Perhaps you've written a package yourself, or you want to use a package that is only available as source code on GitHub or a similar website. There are multiple way to approach this, most of them roughly equivalent. The first step for all of them is to write a `setup.py` file for your package. Some instructions for writing a `setup.py` can be found here.

Once you have a working `setup.py`, there are various ways to proceed, in reverse order of complexity:

- Upload your package to PyPI and `pip install <package>` as in previous sections. This is the least flexible because you'll need to upload to PyPI every time your update your package. If you don't own the package, you shouldn't do this!
- Upload your package to a publicly-accessible version control repository and use *pip*'s VCS support to install it (for example, if your package is on GitHub, something like `pip install git+https://github.com/<UserName>/<package>.git`).
- Use the `COPY` build instruction to copy your package directly into the Docker image, then `pip install <path/to/dir/containing/setup.py>` as a `RUN` instruction. Note that your package will need to be in the Docker build context (see the docs for details).

I want to use a base image that doesn't come with Python pre-installed

Say you have an existing Docker image that you need to use (maybe it includes non-Python dependencies that you aren't sure how to install yourself). You need to add Python to this image so that you can run your own code in it. We recommend adding `miniconda` to the image by adding these lines to your Dockerfile:

```
# Dockerfile

# see discussion below
FROM ubuntu:latest
RUN apt-get -y update \
 && apt-get install -y wget
```

(continues on next page)

(continued from previous page)

```
# Docker build arguments
# use the Python version you need
# default to latest version of miniconda (which can then install any version of␣
↪Python)
ARG PYTHON_VERSION=3.6
ARG MINICONDA_VERSION=latest

# set install location, and add the Python in that location to the PATH
ENV CONDA_DIR=/opt/conda
ENV PATH=${CONDA_DIR}/bin:${PATH}

# install miniconda and Python version specified in config
# (and ipython, which is nice for debugging inside the container)
RUN cd /tmp \
 && wget --quiet https://repo.continuum.io/miniconda/Miniconda3-${MINICONDA_
↪VERSION}-Linux-x86_64.sh \
 && bash Miniconda3-${MINICONDA_VERSION}-Linux-x86_64.sh -f -b -p $CONDA_DIR \
 && rm Miniconda3-${MINICONDA_VERSION}-Linux-x86_64.sh \
 && conda install python=${PYTHON_VERSION} \
 && conda clean -y -all
```

After this, you can install HTMap and any other Python packages you need as in the preceeding sections.

Note that in this example we based the image on Ubuntu's base image and installed `wget`, which we used to download the `miniconda` installer. Depending on your base image, you may need to use a different package manager (for example, `yum`) or different command-line file download tool (for example, `curl`).

I want to build an image for use on the Open Science Grid

First, read through OSG's Singularity documentation.

Based on that, our goal will be to build a Docker image and have OSG convert it to a Singularity image that can be served by OSG. The tricky part of this is that Docker's `ENV` instruction won't carry over to Singularity, which is the usual method of etting `python3` on the `PATH` inside the container. To remedy this, we will create a special directory structure that Singularity recognizes and uses to execute instructions with specified environments.

This is not a Singularity tutorial, so the simplest thing to do is copy the entire *singularity.d* directory that *htmap-exec* uses: https://github.com/htcondor/htmap/tree/master/htmap-exec/singularity.d

Anything you need to specify for your environment should be done in `singularity.d/env/90-environment.sh`. This file will be "sourced" (run) when the image starts, before HTMap executes.

In your Dockerfile, you must copy this directory to the correct location inside the image:

```
# Dockerfile snippet

COPY <path/to/singularity.d> /.singularity.d
```

Note the path on the right: a hidden directory at the root of the filesystem. This is just a Singularity convention. The left path is just the location of the `singularity.d` directory you made.

Note that if you `FROM` an `htmap-exec` image, this setup will already be embedded in the image for you.

2.7.2 Output Files

If the "output" of your map function is a file, HTMap's basic functionality will not be sufficient for you. As a toy example, consider a function which takes a string and a number, and writes out a file containing that string repeated that number of times, with a space between each repetition. The file itself will be the output of our function.

```
import htmap

import itertools
from pathlib import Path


@htmap.mapped
def repeat(string, number):
    output_path = Path("repeated.txt")

    with output_path.open(mode="w") as f:
        f.write(" ".join(itertools.repeat(string, number)))
```

This would work great locally, producing a file named `repeated.txt` in the directory we ran the code from. If this same code runs execute-side, the file will still be produced, but HTMap won't know that we care about the file. In fact, the map will appear to be spectacularly useless:

```
with repeat.build_map() as mb:
    mb("foo", 5)
    mb("wiz", 3)
    mb("bam", 2)

repeated = mb.map

print(list(repeated))
# [None, None, None]
```

There's no sign of our output file! (A function with no `return` statement implicitly returns `None`.)

We need to tell HTMap that we are producing an output file. We can do this by adding a call to an HTMap hook function in our mapped function after we create the file:

```
import htmap

import itertools
from pathlib import Path
```

(continues on next page)

(continued from previous page)

```
@htmap.mapped
def repeat(string, number):
    output_path = Path("repeated.txt")

    with output_path.open(mode="w") as f:
        f.write(" ".join(itertools.repeat(string, number)))

    htmap.transfer_output_files(output_path)  # identical, except for this line
```

The *htmap.transfer_output_files()* function tells HTMap to move the files at the given paths back to the submit machine for us. We can then access those files using the *Map.output_files* attribute, which behaves like a sequence indexed by component numbers. The elements of the sequence are `pathlib.Path` pointing to the directories containing the output files from each component, like so:

```
with repeat.build_map() as mb:
    mb("foo", 5)
    mb("wiz", 3)
    mb("bam", 2)

repeated = mb.map

for component, base in enumerate(repeated.output_files):
    path = base / "repeated.txt"
    print(component, path.read_text())

# 0 foo foo foo foo foo
# 1 wiz wiz wiz
# 2 bam bam
```

2.7.3 Transferring Output to Other Places

You may need to transfer output to places that are not the submit machine. HTMap can arrange this for you using the `output_remaps` feature of *MapOptions* in combination with *TransferPath* to specify the destination of the output files.

In the below example, we have a function `move_file` that just tells HTMap to transfer whatever input it is given. We give the path to an input file stored in a S3 bucket named `my-bucket` on some S3 server we can access, with some file (created and placed in the bucket ahead of time) named `in.txt`. Our goal is to get that file back into the bucket, but renamed `out.txt`. To do so, we also create an `output_file` destination, and tell HTMap to "remap" the output transfer via the `output_remaps` argument of *MapOptions*.

```
def move_file(input_path):
    htmap.transfer_output_files(input_path)
```

(continues on next page)

(continued from previous page)

```
bucket = htmap.TransferPath(
    "my-bucket", protocol="s3", location="s3-server.example.com"
)
input_file = bucket / "in.txt"
output_file = bucket / "out.txt"

print(
    input_file
)  # TransferPath(path='my-bucket/in.txt', protocol='s3', location='s3-server.
↪example.com')
print(
    output_file
)  # TransferPath(path='my-bucket/out.txt', protocol='s3', location='s3-server.
↪example.com')

map = htmap.map(
    move_file,
    [input_file],
    map_options=htmap.MapOptions(
        request_memory="128MB",
        request_disk="1GB",
        output_remaps=[{input_file.name: output_file}],
    ),
)
```

After letting the map run, the output file will be in the bucket, and no output file will have been sent back to the submit node (i.e., `m.output_files[0]` will be an empty directory).

2.7.4 Wrapping External Programs

HTMap can only map Python functions, but you might need to call an external program on the execute node. For example, you may need to use a particular Bash utility script, or run a piece of pre-compiled analysis software. In cases like this, the Python standard library's subprocess module can be used to communicate with those programs.

For example, suppose you need to call the Dubious Barology Lyricon (`dbl`) program, a pre-compiled C program that you have stored in your home directory at `~/dbl`. It takes a single integer argument, and "returns" a single integer by printing it to standard output. So a call to `dbl` on the command line looks like

```
$ dbl 4
8
```

To use HTMap with `dbl`, you could write a mapped function that looks something like

```
import subprocess
import htmap


@htmap.mapped(
    map_options=htmap.MapOptions(
        fixed_input_files="dbl",
    )
)
def dbl(x):
    process = subprocess.run(
        ["dbl", str(x)],
        stdout=subprocess.PIPE,  # use capture_output = True in Python 3.7+
    )

    if process.returncode != 0:
        raise Exception("call to dbl failed!")

    return_value = int(process.stdout)

    return return_value
```

You'll need to be careful with functions like this - check for failures in the programs you call, because HTMap will happily return nonsense if the call fails in some strange way. If we do a map, we'll end up with the expected result:

```
result = dbl.map(range(10))

print(list(result))  # [0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

If you want to test this yourself, here's the Dubious Barology Lyricon (really a simple `bash` program):

```
#!/usr/bin/env bash

echo $((2 * $1))
```

If your external program outputs files, you may find the *Output Files* recipe helpful.

2.7.5 Checkpointing Maps

When running on opportunistic resources, HTCondor might "evict" your map components from the execute locations. Evicted components return to the queue and, without your intervention, restart from scratch. However, HTMap can preserve files across an eviction and make them available in the next run. You can use this to write a function that can resume from partial progress when it restarts.

The important thing for you to think about is that **HTMap will always run your function from the start**. That means that the general structure of a checkpointing function should look like this:

```
def function(inputs):
    try:
        ...
        # attempt to reload from a checkpoint file
    except (
        FileNotFoundError,
        ...,
    ):  # catch any errors that indicate that the checkpoint doesn't exist, is␣
↪corrupt, etc.
        # initialize from input data
        ...
    # do work
```

Your work must be written such that it doesn't care where it starts. Generally that means you'll need to replace `for` loops with `while` loops. For example, a simulation that proceeds in 100 steps like this:

```
import htmap


@htmap.mapped
def function(initial_state):
    current_state = initial_state
    for step in range(100):
        current_state = evolve(current_state)

    return current_state
```

would need to become something like

```
import htmap


@htmap.mapped
def function(initial_state):
    try:
        current_step, current_state = load_from_checkpoint(checkpoint_file)
    except FileNotFoundError:
        current_step, current_state = 0, initial_state

    while current_step < 100:
        current_state = evolve(current_state)
        current_step += 1

        if should_write_checkpoint:
            write_checkpoint(current_step, current_state)
            htmap.checkpoint(checkpoint_file)  # important!
```

(continues on next page)

(continued from previous page)

```
    return current_state
```

Note the call to *htmap.checkpoint()*. This function takes the paths to the checkpoint file(s) that you've written and does the necessary behind-the-scenes handling to make them available if the component needs to restart. If you don't call this function, the files will not be available, and your checkpoint won't work!

Concrete Example

Let's work with a more concrete example. Here's the function, along with some code to run it and prove that it checkpointed:

```
from pathlib import Path
import time

import htmap


@htmap.mapped
def counter(num_steps):
    checkpoint_path = Path("checkpoint")
    try:
        step = int(checkpoint_path.read_text())
        print("loaded checkpoint!")
    except FileNotFoundError:
        step = 0
        print("starting from scratch")

    while True:
        time.sleep(1)
        step += 1
        print(f"completed step {step}")

        if step >= num_steps:
            break

        checkpoint_path.write_text(str(step))
        htmap.checkpoint(checkpoint_path)

    return True


map = counter.map([30])

# wait for the component to start
while map.component_statuses[0] is not htmap.ComponentStatus.RUNNING:
```

(continues on next page)

(continued from previous page)

```
    print(map.component_statuses[0])
    time.sleep(1)

# let it run for 10 seconds
print("component has started, letting it run...")
time.sleep(10)

# vacate it (force it off current execute resource)
map.vacate()
print("vacated map")

# wait until it starts up again and finishes
while map.component_statuses[0] is not htmap.ComponentStatus.COMPLETED:
    print(map.component_statuses[0])
    time.sleep(1)

# look at the function output and the stdout from execution
print(map[0])
print(map.stdout(0))
```

The function itself just sleeps for the given amount of time, but it does it in incremental steps so that we can checkpoint its progress. We write checkpoints to a file named `checkpoint` in the current working directory of the script when it executes. We try to load the current step number (stored as text, so we need to convert it to an integer) from that file when we start, and if that fails we start from the beginning. We write a checkpoint after each step, which is overkill (see the next section), but easy to implement for this short example.

The rest of the code (after the function definition) is just there to prove that the example works. If we run this script, we should see something like this:

```
IDLE
# many IDLE messages
IDLE
component has started, letting it run...
vacated map
RUNNING
IDLE
# more IDLE messages
IDLE
RUNNING
# many RUNNING messages
RUNNING
True  # this is map[0]: it's True, not None, so the function finished␣
↪successfully

# a bunch of debug information from the stdout of the component
```

(continues on next page)

(continued from previous page)

```
----- MAP COMPONENT OUTPUT START -----

loaded checkpoint!  # we did it!
completed step 10
completed step 11
completed step 12
completed step 13
completed step 14
completed step 15
completed step 16
completed step 17
completed step 18
completed step 19
completed step 20
completed step 21
completed step 22
completed step 23
completed step 24
completed step 25
completed step 26
completed step 27
completed step 28
completed step 29
completed step 30

-----  MAP COMPONENT OUTPUT END  -----

Finished executing component at 2019-01-20 08:34:31.130818
```

We successfully started from step 10! For a long-running computation, this could represent a significant amount of work. Long-running components on opportunistic resources might be evicted several times during their life, and without checkpointing, may never finish.

Checkpointing Strategy

You generally don't need to write checkpoints very often. We recommend writing a new checkpoint if a certain amount of time has elapsed, perhaps an hour. For example, using the `datetime` library:

```
import datetime

import htmap


def now():
    return datetime.datetime.utcnow()
```

(continues on next page)

(continued from previous page)

```
@htmap.mapped
def function(inputs):
    latest_checkpoint_at = now()

    # load from checkpoint or initialize

    while not_done:
        # do a unit of work

        if now() > latest_checkpoint_at + datetime.timedelta(hours=1):
            # write checkpoint
            latest_checkpoint_at = now()

    return result
```

Caveats

Checkpointing does introduce some complications with HTMap's metadata tracking system. In particular, HTMap only tracks the runtime, stdout, and stderr of the **last execution** of each component. If your components are vacated and start again from a checkpoint, you'll only see the execution time, standard output, and standard error from the second run. If you need that information, you should track it yourself inside your checkpoint files.

2.7.6 Using HTMap on the Open Science Grid

Running HTMap with the Open Science Grid (OSG) requires some special configuration. The OSG does not support Docker, and is also not amenable to HTMap's own Singularity delivery mechanism. However, the OSG does still allow you to run your code inside a Singularity container. The `.htmaprc` file snippet below sets up HTMap to use this support.

```
# .htmaprc

DELIVERY_METHOD = "assume"

[MAP_OPTIONS]
requirements = "HAS_SINGULARITY == TRUE"
"+ProjectName" = "\"<your project name>\""
"+SingularityImage" =␣
↪"\"/cvmfs/singularity.opensciencegrid.org/<repo/tag:version>\""
```

The extra " on the left are to escape the +, which is not normally legal syntax, and the extra \" on the right are to ensure that the actual value is a string.

Note the two places inside < >, where you must supply some information You must specify your OSG project name, and you must specify which OSG-supplied Singularity image to use. For more information on what images are available, see the OSG Singularity documentation. HTMap's own default image, `htmap-exec`, is always available on the OSG. For example, to use `htmap-exec:v0.4.3`, you would set

```
"+SingularityImage" =␣
↪"\"/cvmfs/singularity.opensciencegrid.org/htcondor/htmap-exec:v0.4.3\""
```

For advice on building your own image for the OSG, see *I want to build an image for use on the Open Science Grid*.

CHAPTER

THREE

USING HTCONDOR WITH HTMAP

HTMap is a Python wrapper over the underlying HTCondor API. That means the vast majority of the HTCondor functionality is available. This page is a brief overview of how HTMap uses HTCondor to run your maps. It may be helpful for debugging, or for cross-referencing your HTMap and HTCondor knowledge.

3.1 Component and Job States

Each HTMap map component is represented by an HTCondor job. Map components will usually be in one of four HTCondor job states:

- **Idle**: the job/component has not started running yet; it is waiting to be assigned resources to execute on.
- **Running**: the job/component is running on an execute machine.
- **Held**: HTCondor has decided that it can't run the job/component, but that you (the user) might be able to fix the problem. The job will try to run again if it released.
- **Completed**: the job/component has finished running, and HTMap has collected its output. These jobs will likely leave the HTCondor queue soon.

For more detail, see the relevant HTCondor documentation:

- https://htcondor.readthedocs.io/en/latest/users-manual/managing-a-job.html#checking-on-the-progress-of-jobs
- https://htcondor.readthedocs.io/en/latest/admin-manual/policy-configuration.html#machine-states

3.2 Requesting Resources

The default resources provisioned for your map component can be limiting – what if your job requires more memory or more disk space? HTCondor jobs can request resources, and HTMap supports those requests via *MapOptions*.

MapOptions accepts many of the same keys that condor_submit accepts. Some of the more commonly requested resources are:

- `request_memory`. Possible values are like `"1MB` for 1MB, or `"2GB"` for 2GB of memory.

- `request_cpus`. Possible values are like `"1"` for 1 CPU, or `"2"` for 2 CPUs.
- `request_disk` to request an amount of disk space. Possible values are like `"10GB"` for 10GB, or `"1TB"` for 1 terabyte.

If any of the resource requests are not set, the default values set by your HTCondor cluster administrator will be used.

These would be set with *MapOptions*. For example, this code might be used:

```
options = htmap.MapOptions(
    request_cpus="1",
    request_disk="10GB",
    request_memory="4GB",
)
htmap.map(..., map_options=options)
```

When it's mentioned that "the option `foo` needs to be set" in a submit file, this corresponds to adding the option in the appropriate place in *MapOptions*.

3.3 GPUs

- For any GPU job, the option `request_gpus` needs to be set.
- Many GPU jobs are machine learning jobs. CHTC has a guide on "Run Machine Learning Jobs on the HTC system".

There are some site-specific options. For example, CHTC has a guide on some of these options "Jobs that use GPUs" to run jobs on their GPU Lab. Check with your site's documentation to see if they have any GPU documentation.

3.4 Command Line Tools

HTMap tries to expose a complete interface for submitting and managing jobs, but not for examining the state of your HTCondor pool itself. Here are some HTCondor commands that you may find useful:

- condor_q: seeing the jobs submitted to the scheduler (similar to *htmap.status()*).
- condor_status: seeing resources the different machines have.

The links go an HTML version of the man pages; they are also visible with `man` (e.g., `man condor_q`). Here's a list of possibly useful commands:

```
## See the jobs user foobar has submitted, and their status
condor_q --submitter foobar

## See if how many machines have GPUs, and how many are available
condor_status --constraint "CUDADriverVersion>=10.1" -total
```

(continues on next page)

(continued from previous page)

```
## See the stats on GPU machines (including GPU name)
condor_status -compact -constraint 'TotalGpus > 0' -af Machine TotalGpus␣
↪CUDADeviceName CUDACapability

## See how much CUDA memory on each machine (and how many are available)
condor_status --constraint "CUDADriverVersion>=10.1" -attributes␣
↪CUDAGlobalMemoryMb -json
# See which machines have that much memory
# Also write JSON file so readable by Pandas read_json
condor_status --constraint "CUDADriverVersion>=10.1" -attributes␣
↪CUDAGlobalMemoryMb -attribute Machine -json >> stats.json

## See how many GPUs are available
condor_status --constraint "CUDADriverVersion>=10.1" -total
```

`CUDAGlobalMemoryMb` is not the only attribute that can be displayed; a more complete list is at https://htcondor.readthedocs.io/en/latest/classad-attributes/machine-classad-attributes.html.

CHAPTER

FOUR

TIPS AND TRICKS

4.1 Separate Job Submission/Monitoring/Collection

This is recommended because it's more interactive and more flexible: it doesn't rely on the script being free of bugs on submission. Likewise, un-expected errors can easily be adapted (such as hung jobs, etc). This is most appropriate for medium- or long-running jobs.

The CLI is useful to monitor and modify ongoing jobs. Generally, in simple use cases we recommend writing two or three scripts:

- A script for job submission (which is run once).
- Use the CLI or a script for monitoring jobs (which is run many times).
- A script to collect results (which is a few times).

Each script uses these commands:

- Submission: HTMap's Python API is primarily used here, possibly through *map()*.
- Monitoring: CLI usage is heavy here. `htmap status` is a good way to view a summary. If any of the jobs fail, diagnose why with commands like `htmap reasons` or `htmap errors`.
- Collection: the completed jobs are collected (as mentioned in *How do I only process completed jobs?*) and the results are written to disk/etc.

The CLI is useful for debugging when dealing with component holds and execution errors. It can be used to quickly view the same kind of information as the *Map* API (though we recommend loading up the map in Python once you need to do anything more complex than read text).

4.2 Use the CLI

Use of the CLI is recommended to go alongside separation of submission/monitoring/collection as mentioned above. This section will provide some useful commands.

This command shows the status of each job for various tags:

```
htmap status --live  # See live display of info on each job (and their tags)
```

This might indicate that 4 jobs in tag `foo` are completed and 2 are idle (or waiting to be run).

This command completely deletes the map with tag `foo`, including removing any jobs that are in any state (running, idle, held, whatever). Use this if you want to completely resubmit the map from scratch, without any previous state.

```
htmap remove foo
```

This commands keeps the jobs in the queue, but prevents them from running. This allowed editing them and lets you edit them live.

```
htmap hold foo
```

These commands will show more information about individual maps and map components:

```
htmap logs  # get path to log file; info here is useful for debugging
htmap components foo  # view which component status for tag "foo"
htmap errors foo # view all errors for tag "foo"
htmap stdout foo 0  # view stdout for first component of tag "foo"
htmap stderr foo 0  # view stdout for first component of tag "foo"
htmap reasons foo  # get reasons for holding map "foo"
```

Some of the longer output is useful to pipe into `less` so it's easily navigable and searchable. For example,

```
htmap errors foo | less
```

To get help on `less`, use the command `man less` or press `h` while in `less`.

Full CLI documentation is at *CLI Reference*.

4.3 Conditional Execution on Cluster vs. Submit

The environment variable `HTMAP_ON_EXECUTE` is set to `'1'` while map components are executing out on the cluster. This can be useful if you need to switch certain behavior on or off depending whether you're running your function locally or not.

4.4 Functional programming

4.4.1 Filter

In the parlance of higher-order functions, HTMap only provides map. Another higher-order function, filter, is easy to implement once you have a map. To mimic it we create a map with a boolean output, and use *htmap.Map.iter_with_inputs()* inside a list comprehension to filter the inputs using the outputs.

Here's a brief example: checking whether integers are even.

```
import htmap


@htmap.mapped
def is_even(x: int) -> bool:
    return x % 2 == 0


result = is_even.map(range(10))

filtered = [input for input, output in result.iter_with_inputs() if output]

print(filtered)  # [((0,), {}), ((2,), {}), ((4,), {}), ((6,), {}), ((8,), {})]
```

4.4.2 Groupby

In the parlance of higher-order functions, HTMap only provides map. Another higher-order function, groupby, is easy to implement once you have a map. To mimic it we'll write a helper function that uses a `collections.defaultdict` to construct a dictionary that collects inputs that have the same output, using the output as the key.

Here's a brief example: grouping integer by whether they are even or not.

```
import collections
import htmap


@htmap.mapped
def is_even(x: int) -> bool:
    return x % 2 == 0


def groupby(result):
    groups = collections.defaultdict(list)

    for input, output in result.iter_with_inputs():
        groups[output].append(input)

    return groups


result = is_even.map(range(10))

for group, elements in groupby(result).items():
    print(group, elements)
```

(continues on next page)

(continued from previous page)

```
# True [((0,), {}), ((2,), {}), ((4,), {}), ((6,), {}), ((8,), {})]
# False [((1,), {}), ((3,), {}), ((5,), {}), ((7,), {}), ((9,), {})]
```

CHAPTER

FIVE

FAQ

5.1 How do I abort a job?

For example, say you mistakenly launched a map tagged `foo`, but now want to abort/cancel it, fix some input parameters, then relaunch it.

The right CLI command is `htmap remove foo`, or the HTMap function *remove()*. This mirrors the HTCondor API and will remove the job from the job scheduler regardless of state (running, idle, held, etc).

5.2 How do I only process completed jobs?

Let's say you submitted 10,000 long-running jobs, and 99.9% of these jobs have finished successfully. You'd like to get the results from the successful jobs, and save the results to disk without have to wait for the 10 remaining jobs slow jobs.

The right function to use is *components_by_status()*. It can filter out the successful jobs and process those. See the *components_by_status()* documentation for an example usage.

5.3 Is it possible to use Dask with HTCondor? How does it compare with HTMap?

HTMap provides a transparent interface to the underlying HTCondor behavior, allowing for features like using HTCondor file transfer and taking advantage of the rich HTCondor job model. HTMap does need to be running through the entire duration of your computation.

Dask can spawn its distributed workers on an HTCondor pool. By doing this you get access to Dask's features, but not HTCondor's. Dask will need to be running through the entire duration of your computation.

You should choose the appropriate option for your use case.

Dask Distributed is a lightweight library for distributed Python computation. Dask Distributed has familiar APIs, is declarative and supports more complex scheduling than map/filter/reduce.

Dask-Jobqueue present a wrapper for HTCondor clusters through their HTCondorCluster. After HTCondorCluster is used, Dask can be used as normal or on your own machine. This is common with other cluster

managers too: Dask-Jobqueue also wraps SLURM, SGE, PBS and LSF clusters, and Dask Distributed can wrap Kubernetes and Hadoop clusters.

5.4 I'm getting a weird error from `cloudpickle.load`?

You probably have a version mismatch between the submit and execute locations. See the "Attention" box near the top of *Dependency Management*.

If you are using custom libraries, always import them before trying to load any output from maps that use them.

5.5 I'm getting an error about a job being held. What should I do?

Your code likely encountered an error during remote execution. Briefly, try viewing the standard error (`stderr`) with HTMap, either via the CLI or API. Details can be found in *Tutorials* and *Error Handling*.

CHAPTER

SIX

API REFERENCE

6.1 Tags and Map Persistence

The `tag` is the central organizing piece of data in HTMap. Every map that you run produces a *Map* which is connected to a unique `tag`. A `tag` cannot be re-used until the associated map has been deleted or retagged. You can either provide a tag or let HTMap generate one automatically.

If you do not provide a tag, the map will be marked as **transient**. Transient maps will be removed by the *htmap.clean()* function without passing `all = True`, while non-transient (i.e., persistent) maps will not. If you provide a tag during map creation or *htmap.Map.retag()* a map, it will be marked as persistent.

6.2 Mapping Functions

htmap.**map**(*func*, *args*, *map_options=None*, *tag=None*, *quiet=False*)

Map a function call over a one-dimensional iterable of arguments. The function must take exactly one positional argument and no keyword arguments.

Parameters

- **func** (Callable) – The function to map the arguments over.
- **args** (Iterable[Any]) – An iterable of arguments to pass to the mapped function.
- **map_options** (Optional[*MapOptions*]) – An instance of *htmap.MapOptions*.
- **tag** (Optional[str]) – The `tag` to assign to this map.
- **quiet** (bool) – Do not print the map name in an interactive shell.

Return type

Map

Returns

map – A *htmap.Map* representing the map.

htmap.**starmap**(*func*, *args=None*, *kwargs=None*, *map_options=None*, *tag=None*, *quiet=False*)

Map a function call over aligned iterables of arguments and keyword arguments. Each element of `args` and `kwargs` is unpacked into the signature of the function, so their elements should be tuples and dictionaries corresponding to position and keyword arguments of the mapped function.

Parameters

- **func** (Callable) – The function to map the arguments over.
- **args** (Optional[Iterable[Tuple[Any, ...]]]) – An iterable of tuples of positional arguments to unpack into the mapped function.
- **kwargs** (Optional[Iterable[Dict[str, Any]]]) – An iterable of dictionaries of keyword arguments to unpack into the mapped function.
- **map_options** (Optional[*MapOptions*]) – An instance of *htmap.MapOptions*.
- **tag** (Optional[str]) – The `tag` to assign to this map.
- **quiet** (bool) – Do not print the map name in an interactive shell.

Return type
Map

Returns
map – A *htmap.Map* representing the map.

htmap.**build_map**(*func*, *map_options=None*, *tag=None*)

Return a *MapBuilder* for the given function.

Parameters

- **func** (Callable) – The function to map over.
- **map_options** (Optional[*MapOptions*]) – An instance of *htmap.MapOptions*.
- **tag** (Optional[str]) – The `tag` to assign to this map.

Return type
MapBuilder

Returns
map_builder – A *MapBuilder* for the given function.

6.3 Map Builder

class htmap.**MapBuilder**(*func*, *map_options=None*, *tag=None*)

The *htmap.MapBuilder* provides an alternate way to create maps. Once created via *htmap.build_map()* or similar as a context manager, the map builder can be called as if it were the function you're mapping over. When the `with` block exits, the inputs are collected and submitted as a single map.

```
with htmap.build_map(tag="pow", func=lambda x, p: x ** p) as builder:
    for x in range(1, 4):
        builder(x, x)

map = builder.map
print(list(map))  # [1, 4, 27]
```

__call__(**args*, ***kwargs*)

Adds the given inputs to the map.

Return type
None

__len__()

The length of a *MapBuilder* is the number of inputs it has been sent.

Return type
int

property map: *Map*

The *Map* associated with this *MapBuilder*. Will raise *htmap.exceptions.NoMapYet* when accessed until the `with` block for this *MapBuilder* completes.

6.4 MappedFunction

A more convenient and flexible way to work with HTMap is to use the `htmap()` decorator to build a *MappedFunction*.

htmap.**mapped**(*map_options=None*)

A decorator that wraps a function in an *MappedFunction*, which provides an interface for mapping functions calls out to an HTCondor cluster.

Parameters
map_options (Optional[*MapOptions*]) – An instance of *htmap.MapOptions*. Any map calls from the *MappedFunction* produced by this decorator will inherit from this.

Return type
Union[Callable, *MappedFunction*]

Returns
mapped_function – A *MappedFunction* that wraps the function (or a wrapper function that does the wrapping).

class htmap.**MappedFunction**(*func*, *map_options=None*)

Parameters

- **func** (Callable) – A function to wrap in a *MappedFunction*.
- **map_options** (Optional[*MapOptions*]) – An instance of *htmap.MapOptions*. Any map calls from the *MappedFunction* produced by this decorator will inherit from this.

map(*args*, *tag=None*, *map_options=None*)

As *htmap.map()*, but the `func` argument is the mapped function.

Return type
Map

starmap(*args=None*, *kwargs=None*, *tag=None*, *map_options=None*)

As *htmap.starmap()*, but the `func` argument is the mapped function.

Return type
Map

build_map(*tag=None*, *map_options=None*)

As *htmap.build_map()*, but the `func` argument is the mapped function.

Return type
MapBuilder

6.5 Map

The *Map* is your window into the status and output of your map. Once you get a map result back from a map call, you can use its methods to get the status of jobs, change the properties of the map while its running, pause, restart, or cancel the map, and finally retrieve the output once the map is done.

The various methods that allow you to get and iterate over components will raise exceptions if something has gone wrong with your map:

- *htmap.exceptions.MapComponentError* if a component experienced an error while executing.
- *htmap.exceptions.MapComponentHeld* if a component was held by HTCondor, likely because an input file did not exist or the component used too much memory or disk.

The exception message will contain information about what caused the error. See *Error Handling* for more details on error handling.

class htmap.**Map**(*, *tag*, *map_dir*)

Represents the results from a map call.

> **Warning:** You should never instantiate a *Map* directly! Instead, you'll get your *Map* by calling a top-level mapping function like *htmap.map()*, a *MappedFunction* mapping method, or by using *htmap.load()*. We are not responsible for whatever vile contraption you build if you bypass the correct methods!

__len__()

The length of a *Map* is the number of components it contains.

__getitem__(*item*)

Return the output associated with the input index. Does not block.

Return type
Any

classmethod load(*tag*)

Load a *Map* by looking up its `tag`.

Raises *htmap.exceptions.TagNotFound* if the `tag` does not exist.

Parameters
tag (str) – The tag to search for.

Return type
Map

Returns
map – The map with the given tag.

property components: Tuple[int, ...]

Return a tuple containing the component indices for the *htmap.Map*.

property is_done: bool

True if all of the output is available for this map.

property is_active: bool

True if any map components are not complete (or errored!).

wait(*timeout=None*, *show_progress_bar=False*, *holds_ok=False*, *errors_ok=False*)

Wait until all output associated with this *Map* is available.

If any components in the map are held or experience an execution error, this method will raise an exception (*htmap.exceptions.MapComponentHeld* or *htmap.exceptions.MapComponentError*, respectively).

Parameters

- **timeout** (Union[int, float, timedelta, None]) – How long to wait for the map to complete before raising a *htmap.exceptions.TimeoutError*. If None, wait forever.
- **show_progress_bar** (bool) – If True, a progress bar will be displayed.
- **holds_ok** (bool) – If True, will not raise exceptions if components are held.
- **errors_ok** (bool) – If True, will not raise exceptions if components experience execution errors.

Return type
None

get(*component*, *timeout=None*)

Return the output associated with the input component index. If the component experienced an execution error, this will raise *htmap.exceptions.MapComponentError*. Use *get_err()*, *errors()*, *error_reports()* to see what went wrong!

Parameters

- **component** (int) – The index of the input to get the output for.
- **timeout** (Union[int, float, timedelta, None]) – How long to wait for the output to exist before raising a *htmap.exceptions.TimeoutError*. If None, wait forever.

Return type
Any

get_err(*component*, *timeout=None*)

Return the error associated with the input component index. If the component actually succeeded, this will raise *htmap.exceptions.ExpectedError*.

Parameters

- **component** (int) – The index of the input to get the output for.
- **timeout** (Union[int, float, timedelta, None]) – How long to wait for the output to exist before raising a *htmap.exceptions.TimeoutError*. If None, wait forever.

Return type

ComponentError

iter(*timeout=None*)

Returns an iterator over the output of the *htmap.Map* in the same order as the inputs, waiting on each individual output to become available.

Parameters

timeout (Union[int, float, timedelta, None]) – How long to wait for each output to be available before raising a *htmap.exceptions.TimeoutError*. If None, wait forever.

Return type

Iterator[Any]

iter_with_inputs(*timeout=None*)

Returns an iterator over the inputs and output of the *htmap.Map* in the same order as the inputs, waiting on each individual output to become available.

Parameters

timeout (Union[int, float, timedelta, None]) – How long to wait for each output to be available before raising a *htmap.exceptions.TimeoutError*. If None, wait forever.

Return type

Iterator[Tuple[Tuple[tuple, Dict[str, Any]], Any]]

iter_as_available(*timeout=None*)

Returns an iterator over the output of the *htmap.Map*, yielding individual outputs as they become available.

The iteration order is initially random, but is consistent within a single interpreter session once the map is completed.

Parameters

timeout (Union[int, float, timedelta, None]) – How long to wait for the entire iteration to complete before raising a *htmap.exceptions.TimeoutError*. If None, wait forever.

Return type

Iterator[Any]

iter_as_available_with_inputs(*timeout=None*)

Returns an iterator over the inputs and output of the *htmap.Map*, yielding individual `(input, output)` pairs as they become available.

The iteration order is initially random, but is consistent within a single interpreter session once the map is completed.

Parameters

timeout (Union[int, float, timedelta, None]) – How long to wait for the entire iteration to complete before raising a *htmap.exceptions.TimeoutError*. If None, wait forever.

Return type

Iterator[Tuple[Tuple[tuple, Dict[str, Any]], Any]]

iter_inputs()

Returns an iterator over the inputs of the *htmap.Map*.

Return type

Iterator[Any]

property component_statuses: List[*ComponentStatus*]

Return the current *state.ComponentStatus* of each component in the map.

components_by_status()

Return the component indices grouped by their states.

Return type

Mapping[*ComponentStatus*, Tuple[int, ...]]

Examples

This example finds the completed jobs for a submitted map, and processes those results:

```
from time import sleep
import htmap

def job(x):
    sleep(x)
    return 1 / x


m = htmap.map(job, [0, 2, 4, 6, 8], tag="foo")

# Wait for all jobs to finish.
# Alternatively, use `futures = htmap.load("foo")` on a different␣
↪process
sleep(10)

completed = m.components_by_status()[htmap.JobStatus.COMPLETED]
```

(continues on next page)

(continued from previous page)

```
for component in completed:
    result = m.get(future)
    # Whatever processing needs to be done
    print(result)  # prints "2", "4", "6", and "8"
```

status()

Return a string containing the number of jobs in each status.

Return type
str

property holds: Dict[int, *ComponentHold*]

A dictionary of component indices to their `Hold` (if they are held).

hold_report()

Return a string containing a formatted table describing any held components.

Return type
str

property errors: Dict[int, *ComponentError*]

A dictionary of component indices to their `ExecutionError` (if that component experienced an error).

error_reports()

Yields the error reports for any components that experienced an error during execution.

Return type
Iterator[str]

property memory_usage: List[int]

Return the latest peak memory usage of each map component, measured in MB. A component that hasn't reported yet will show a `0`.

> **Warning:** Due to current limitations in HTCondor, memory use for very short-lived components (<5 seconds) will not be accurate.

property runtime: List[timedelta]

Return the total runtime (user + system) of each component.

property local_data: int

Return the number of bytes stored on the local disk by the map.

remove(*force=False*)

This command removes a map from the Condor queue. Functionally, this command aborts a job.

This function will completely remove a map from the Condor queue regardless of job state (running, executing, waiting, etc). All data associated with a removed map is permanently deleted.

Parameters
force (bool) – If True, do not wait for HTCondor to remove the map components before removing local data.

Return type
None

property exists: bool

True if and only if the map has **not** been successfully removed. Otherwise, False.

hold()

This command holds a map. The components of the map will not be allowed to run until released (see *Map.release()*).

HTCondor may itself hold your map components if it detects that something has gone wrong with them. Resolve the underlying problem, then use the *Map.release()* command to allow the components to run again.

Return type
None

release()

This command releases a map, undoing holds (see *Map.hold()*). The held components of a released map will become idle again.

HTCondor may itself hold your map components if it detects that something has gone wrong with them. Resolve the underlying problem, then use this command to allow the components to run again.

Return type
None

pause()

This command pauses a map. The running components of a paused map will keep their resource claims, but will stop actively executing. The map can be un-paused by resuming it (see the *Map.resume()* command).

Return type
None

resume()

This command resumes a map (reverses the *Map.pause()* command). The running components of a resumed map will resume execution on their claimed resources.

Return type
None

vacate()

This command vacates a map. The running components of a vacated map will give up their claimed resources and become idle again.

Checkpointing maps will still have access to their last checkpoint, and will resume from it as if execution was interrupted for any other reason.

Return type
None

`set_memory`(*memory*)

Change the amount of memory (RAM) each map component needs.

> **Warning:** Edits do not affect components that are currently running. To "restart" components so that they see the new attribute value, consider vacating their map (see the vacate command).

Parameters
memory (int) – The amount of memory (RAM) to request, as an integer number of MB.

Return type
None

`set_disk`(*disk*)

Change the amount of disk space each map component needs.

> **Warning:** Edits do not affect components that are currently running. To "restart" components so that they see the new attribute value, consider vacating their map (see the vacate command).

Parameters
disk (int) – The amount of disk space to request, as an integer number of KB.

Return type
None

`rerun`(*components=None*)

Re-run (part of) the map from scratch. The selected components must be completed or errored.

Any existing output of re-run components is removed; they are re-submitted to the HTCondor queue with their original map options (i.e., without any subsequent edits).

Parameters
components (Optional[Iterable[int]]) – The components to rerun. If None, the entire map will be re-run.

Return type
None

`retag`(*tag*)

Give this map a new `tag`. The old `tag` will be available for re-use immediately.

Retagging a map makes it not transient. Maps that have never had an explicit tag given to them are transient and can be easily cleaned up via the clean command.

Parameters
tag (str) – The `tag` to assign to the map.

Return type
None

property is_transient: bool

`True` is the map is transient, `False` otherwise.

property stdout: *MapStdOut*

A sequence containing the `stdout` for each map component. You can index into it (with a component index) to get the `stdout` for that component, or iterate over the sequence to get all of the `stdout` from the map.

property stderr: *MapStdErr*

A sequence containing the `stderr` for each map component. You can index into it (with a component index) to get the `stderr` for that component, or iterate over the sequence to get all of the `stderr` from the map.

property output_files: *MapOutputFiles*

A sequence containing the path to the directory containing the output files for each map component. You can index into it (with a component index) to get the path for that component, or iterate over the sequence to get all of the paths from the map.

count(*value*) → integer -- return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer -- return first index of value.

Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

class htmap.**ComponentStatus**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

An enumeration of the possible statuses that a map component can be in. These are mostly identical to the HTCondor job statuses of the same name.

UNKNOWN = 'UNKNOWN'

UNMATERIALIZED = 'UNMATERIALIZED'

IDLE = 'IDLE'

RUNNING = 'RUNNING'

REMOVED = 'REMOVED'

COMPLETED = 'COMPLETED'

HELD = 'HELD'

SUSPENDED = 'SUSPENDED'

ERRORED = 'ERRORED'

classmethod display_statuses()

Return type
Tuple[*ComponentStatus*, ...]

class htmap.**MapStdOut**(*map*)

An object that helps implement a map's sequence over its `stdout`. Don't both instantiating one yourself: use the `Map.stdout` attribute instead.

get(*component*, *timeout=None*)

Return a string containing the stdout/stderr from a single map component.

Parameters

- **component** (int) – The index of the map component to look up.
- **timeout** (Union[int, float, timedelta, None]) – How long to wait before raising a *htmap.exceptions.TimeoutError*. If None, wait forever.

Return type
str

Returns
stdx – The standard output/error of the map component.

class htmap.**MapStdErr**(*map*)

An object that helps implement a map's sequence over its `stderr`. Don't both instantiating one yourself: use the `Map.stderr` attribute instead.

get(*component*, *timeout=None*)

Return a string containing the stdout/stderr from a single map component.

Parameters

- **component** (int) – The index of the map component to look up.
- **timeout** (Union[int, float, timedelta, None]) – How long to wait before raising a *htmap.exceptions.TimeoutError*. If None, wait forever.

Return type
str

Returns
stdx – The standard output/error of the map component.

class htmap.**MapOutputFiles**(*map*)

An object that helps implement a map's sequence over its output file directories. Don't both instantiating one yourself: use the `Map.output_files` attribute instead.

get(*component*, *timeout=None*)

Return the pathlib.Path to the directory containing the output files for the given component.

Parameters

- **component** (int) – The index of the map component to look up.
- **timeout** (Union[int, float, timedelta, None]) – How long to wait before raising a *htmap.exceptions.TimeoutError*. If None, wait forever.

Return type
Path

Returns
path – The path to the directory containing the output files for the given component.

6.6 Error Handling

Map components can generally encounter two kinds of errors:

- An exception occurred inside your function on the execute node.
- HTCondor was unable to run the map component for some reason.

The first kind will result in HTMap transporting a *htmap.ComponentError* back to you, which you can access via *htmap.Map.get_err()*. The *htmap.ComponentError.report()* method returns a formatted error report for your perusal. *htmap.Map.error_reports()* is a shortcut that returns all of the error reports for all of the components of your map. If you want to access the error programmatically, you can grab it using `htmap.get_err()`.

The second kind of error doesn't provide as much information. The method *htmap.Map.holds()* will give you a dictionary mapping components to their *htmap.ComponentHold*, if they have one. *htmap.Map.hold_report()* will return a formatted table showing any holds in your map. The hold's `reason` attribute will tell you a lot about what HTCondor doesn't like about your component.

class htmap.**ComponentError**(*, *map*, *component*, *exception_msg*, *node_info*, *python_info*, *scratch_dir_contents*, *stack_summary*)

Represents an error experienced by a map component during remote execution.

map

The *htmap.Map* the component is a part of.

Type
htmap.Map

component

The component index from the map.

Type
int

exception_msg

The raw message string from the remote exception.

Type
str

node_info

A tuple containing information about the HTCondor execute node the component ran on.

Type
tuple

python_info

A tuple containing information about the Python installation on the execute node.

Type
tuple

scratch_dir_contents

A list of paths in the scratch directory on the execute node.

Type
List[pathlib.Path]

stack_summary

The Python stack frames at the time of execution, excluding HTMap's own stack frame.

Type
traceback.StackSummary

report()

Return a formatted error report.

The raw information in this report is available in the attributes of this class.

Return type
str

class htmap.**ComponentHold**(*code*, *reason*)

Represents an HTCondor hold on a map component.

Parameters

- **code** (int) – The HTCondor HoldReasonCode.
- **reason** (str) – The HTCondor hold reason.

6.7 MapOptions

Map options are the equivalent of HTCondor's submit descriptors. All HTCondor submit descriptors are valid map options **except** those reserved by HTMap for internal use (see below).

Fixed options are the most basic option. The entire map will used the fixed option. If you pass a single string as the value of a map option, it will become a fixed option.

Variadic options are options that are given individually to each component of a map. For example, each component of a map might need a different amount of memory. In that case you could pass a list to request_memory, with the same number of elements as the number of inputs to the map.

Inherited options are given to a *htmap.MappedFunction* when it is created. Any maps made using that function can inherit these options. Options that are passed in the actual map call override inherited options (excepting `fixed_input_files`, see the note). For example, if you know that a certain function always takes a large amount of memory, you could give it a large `request_memory` at the *htmap.MappedFunction* level so that you don't have to do it for every individual map. Additionally, default map options can be set globally via `settings['MAP_OPTIONS.<option_name>'] = <option_value>`.

Warning: Only certain options make sense as inherited options. For example, they shouldn't be variadic options.

`fixed_input_files` has special behavior as an inherited option: they are *merged together* instead of overridden.

Note: When looking at examples of raw HTCondor submit files, you may see submit descriptors that are prefixed with a + or a `MY.`. Those options should be passed to *htmap.MapOptions* via the `custom_options` keyword arguments.

class htmap.**MapOptions**(*, *fixed_input_files=None*, *input_files=None*, *output_remaps=None*, *custom_options=None*, ***kwargs*)

Parameters

- **fixed_input_files** (Union[PathLike, *TransferPath*, Iterable[Union[PathLike, *TransferPath*]], None]) – A single file, or an iterable of files, to send to all components of the map.
- **input_files** (Union[Iterable[Union[PathLike, *TransferPath*]], Iterable[Iterable[Union[PathLike, *TransferPath*]]], None]) – An iterable of single files or iterables of files to map over. This may be useful if you want additional files to be sent to each map component, but don't want them in your mapped function's arguments.
- **output_remaps** (Union[Mapping[str, *TransferPath*], Iterable[Mapping[str, *TransferPath*]], None]) – A dictionary, or an iterable of dictionaries, specifying output transfer remaps. A remapped output file is sent to a specified destination instead of back to the submit machine. If a single dictionary is passed, it will be applied to every map component (in this case, you may want to use the `$(component)` submit macro to differentiate them). Each dictionary should be a "mapping" between the **names** (last path component, as a string) of o utput files and their **destinations**, given as a *TransferPath*. You must still call *transfer_output_files()* on the files for the them to be transferred at all; listing them here *only* sets up the remapping.
- **custom_options** (Optional[Dict[str, str]]) – A dictionary of submit descriptors that are *not* built-in HTCondor descriptors. These are the descriptors that, if you were writing a submit file, would have a leading + or `MY.`. The leading characters are unnecessary here, but can be included if you'd like.

- **kwargs** (Union[str, Iterable[str]]) – Additional keyword arguments are interpreted as HTCondor submit descriptors. Values that are single strings are used for all components of the map. Providing an iterable for the value will map that option. Certain keywords are reserved for internal use (see the RESERVED_KEYS class attribute).

Notes

Warning: The representation of the values in `fixed_input_files`, `input_files`, `custom_options` and `kwargs` should exactly match the characters in the submit file after the `=`.

For example, let's say your job requires this submit file:

```
# file: job.submit
foo = "bar"
aaa = xyz
bbb = false
ccc = 1
```

The `MapOptions` that express the same submit options would be:

```
>>> options = {"foo": '"bar"', "aaa": "xyz", "bbb": "false", "ccc": "1"}
>>> print(options["foo"])  # exactly matches the value in the submit file
... "bar"
>>> options["foo"] = "\"bar\""  # alternative value
>>> MapOptions(**options)
```

Submit file values with quotes require escaped quotes in the Python string.

RESERVED_KEYS = {'+IsHTMapJob', '+component', 'IsHTMapJob', 'MY.IsHTMapJob', 'MY.component', 'arguments', 'component', 'executable', 'jobbatchname', 'log', 'should_transfer_files', 'stderr', 'stdout', 'submit_event_notes', 'transfer_executable', 'transfer_input_files', 'transfer_output_files', 'transfer_output_remaps', 'universe', 'when_to_transfer_output'}

classmethod merge(**others*)

Merge any number of *MapOptions* together, like a `collections.ChainMap`. Options closer to the left take priority. :rtype: *MapOptions*

Note: `fixed_input_files` is a special case, and is merged up the chain instead of being overwritten. `requirements` are also combined, in a way where all requirements must be satisfied.

6.8 File Transfer

class htmap.**TransferPath**(*path*, *protocol=None*, *location=None*)

A *TransferPath* describes the location of a file or directory. If the `protocol` and `location` are both `None`, it describes a location on the local filesystem. If either are given, it describes a remote location.

When used as an argument to a mapped function, a *TransferPath* tells HTMap to arrange for the specified files/directories to be transferred to the execute machine from some location, which may be the local filesystem on the submit machine or some remote location like an HTTP address or an S3 server.

Transfer paths are recognized in mapped function inputs as long as they are either:

1. Arguments or keyword arguments of the mapped function.
2. Stored inside a primitive container (tuple, list, set, dictionary value) that is an argument or keyword argument of the mapped function. Nested containers are inspected recursively.

When the mapped function runs execute-side, it will receive (instead of this object) a normal `pathlib.Path` object pointing to the execute-side path of the file/directory.

TransferPath is also used to specify the locations for output files to be sent, if they are not to be returned to the submit machine. For example, output files could be sent to an S3 server. See the `output_remaps` argument of *MapOptions* for more details on "remapped" output file transfer.

Where appropriate, *TransferPath* has the same interface as a `pathlib.Path`. See the examples for some ways to leverage this API to efficiently construct transfer paths.

> **Attention:** You may need to pass additional submit descriptors to your map to actually be able to use input/output transfers for certain protocols. For example, to transfer to and from an S3 server, you also need to pass `aws_access_key_id_file` and `aws_secret_access_key_file`. See the condor_submit documentation for more details.

Examples

Transfer a file stored in your home directory using HTCondor file transfer:

```
transfer_path = htmap.TransferPath.cwd() / 'file.txt'
```

Transfer a local file at an absolute path using HTCondor file transfer:

```
transfer_path = htmap.TransferPath("/foo/bar/baz.txt")
```

Get a file from an HTTP server, located at `http://htmap.readthedocs.io/en/latest/_static/htmap-logo.svg`:

```
transfer_path = htmap.TransferPath(
    path = "en/latest/_static/htmap-logo.svg",
    protocol = "http",
    location = "htmap.readthedocs.io",
)
```

or

```
base_path = htmap.TransferPath(
    path = "/",
    protocol = "http",
    location = "htmap.readthedocs.io",
)
transfer_path = base_path / 'en' / 'latest' / '_static' / 'htmap-logo.svg'
```

Parameters

- **path** (Union[*TransferPath*, PathLike]) – The path to the file or directory to transfer.
- **protocol** (Optional[str]) – The protocol to perform for the transfer with. If set to None (the default), use HTCondor local file transfer.
- **location** (Optional[str]) – The location to find a remote file when using a protocol transfer. This could be the address of a server, for example.

htmap.**transfer_output_files**(**paths*)

Informs HTMap about the existence of output files.

> **Attention:** This function is a no-op when executing locally, so you if you're testing your function it won't do anything.

> **Attention:** The files will be **moved** by this function, so they will not be available in their original locations.

Parameters

paths (PathLike) – The paths to the output files.

Return type

None

6.9 Checkpointing

htmap.**checkpoint**(**paths*)

Informs HTMap about the existence of checkpoint files. This function should be called every time the checkpoint files are changed, even if they have the same names as before.

> **Attention:** This function is a no-op when executing locally (i.e., not execute-side), so you if you're testing your function locally it won't do anything.

> **Attention:** The files will be **copied** by this function, so try not to make the checkpoint files too large.

Parameters

paths (PathLike) – The paths to the checkpoint files.

Return type

None

6.10 Management

These functions help you manage your maps.

htmap.**status**(*maps=None*, *include_state=True*, *include_meta=True*)

Return a formatted table containing information on the given maps.

Parameters

- **maps** (Optional[Iterable[*Map*]]) – The maps to display information on. If None, displays information on all existing maps.
- **include_state** (bool) – If True, include information on the state of the map's components.
- **include_meta** (bool) – If True, include information about the map's memory usage, disk usage, and runtime.

Return type

str

Returns

table – A text table containing information on the given maps.

htmap.**get_tags**(*pattern=None*)

Return a tuple containing the tag for all existing maps, with optional filtering based on a glob-style pattern.

Parameters
pattern (Optional[str]) – A glob-style pattern. Only tags that fit the pattern will be returned. If None (the default), all tags will be returned.

Return type
Tuple[str, ...]

Returns
tags – A tuple containing the tags that match the pattern.

htmap.**load**(*tag*)

Reconstruct a *Map* from its tag.

Parameters
tag (str) – The tag to search for.

Return type
Map

Returns
map – The result with the given tag.

htmap.**load_maps**(*pattern=None*)

Return a tuple containing the *Map* for all existing maps, with optional filtering based on a glob-style pattern.

Parameters
pattern (Optional[str]) – A glob-style pattern. Only maps whose tags fit the pattern will be returned. If None (the default), all maps will be returned.

Return type
Tuple[*Map*, ...]

Returns
maps – A tuple contain the maps whose tags fit the pattern.

htmap.**remove**(*tag*, *not_exist_ok=True*)

Remove the map with the given tag.

Parameters

- **tag** (str) – The tag to search for and remove.
- **not_exist_ok** (bool) – If False, raise htmap.exceptions.MapIdNotFound if the tag doesn't exist.

Return type
None

htmap.**clean**(*, *all=False*)

Clean up transient maps by removing them.

Maps that have never had a tag explicitly set are assigned randomized tags and marked as "transient". This command removes maps marked transient (and can also remove all maps, not just transient ones, if the –all option is passed).

Parameters
all (bool) – If True, remove all maps, not just transient ones. Defaults to False.

Return type
List[str]

Returns
cleaned_tags – A list of the tags of the maps that were removed.

6.10.1 Programmatic Status Messages

These functions are useful for generating machine-readable status information.

htmap.**status_json**(*maps=None*, *include_state=True*, *include_meta=True*, *compact=False*)

Return a JSON-formatted string containing information on the given maps.

Disk and memory usage are reported in bytes. Runtimes are reported in seconds.

Parameters

- **maps** (Optional[Iterable[*Map*]]) – The maps to display information on. If None, displays information on all existing maps.
- **include_state** (bool) – If True, include information on the state of the map's components.
- **include_meta** (bool) – If True, include information about the map's memory usage, disk usage, and runtime.
- **compact** (bool) – If True, the JSON will be formatted in the most compact possible representation.

Return type
str

Returns
json – A JSON-formatted dictionary containing information on the given maps.

htmap.**status_csv**(*maps=None*, *include_state=True*, *include_meta=True*)

Return a CSV-formatted string containing information on the given maps.

Disk and memory usage are reported in bytes. Runtimes are reported in seconds.

Parameters

- **maps** (Optional[Iterable[*Map*]]) – The maps to display information on. If None, displays information on all existing maps.
- **include_state** (bool) – If True, include information on the state of the map's components.
- **include_meta** (bool) – If True, include information about the map's memory usage, disk usage, and runtime.

Return type
str

Returns

csv – A CSV-formatted table containing information on the given maps.

6.11 Delivery Methods

htmap.**register_delivery_method**(*name*, *descriptors_func*, *setup_func=None*)

Register a new delivery method with HTMap.

Parameters

- **name** (str) – The name of the delivery method; this is what the DELIVERY_METHOD should be set to to use this delivery method.
- **descriptors_func** (Callable[[str, Path], dict]) – The function that provides the HTCondor submit descriptors for this delivery method.
- **setup_func** (Optional[Callable[[str, Path], None]]) – The function that does any setup necessary to running the map.

Return type

None

6.11.1 Transplant Installs

These functions help you manage your transplant installs.

htmap.**transplants**()

Return type

Tuple[*Transplant*, ...]

class htmap.**Transplant**(*hash: str*, *path: Path*, *created: datetime*, *size: int*, *packages: Tuple[str, ...]*)

An object that represents metadata information about a transplant install.

Create new instance of Transplant(hash, path, created, size, packages)

hash: str

Alias for field number 0

path: Path

Alias for field number 1

created: datetime

Alias for field number 2

size: int

Alias for field number 3

packages: Tuple[str, ...]

Alias for field number 4

classmethod load(*path*)

Parameters
path (Path) – The path to the transplant install.

Return type
Transplant

Returns
transplant – The *Transplant* that represents the transplant install.

remove()

htmap.**transplant_info**()

Return type
str

6.12 Settings

HTMap exposes configurable settings through `htmap.settings`, which is an instance of the class *htmap.settings.Settings*. This settings object manages a lookup chain of dictionaries. The settings object created during startup contains two dictionaries. The lowest level contains HTMap's default settings, and the next level up is constructed from a settings file at `~/.htmaprc`. If that file does not exist, an empty dictionary is used instead. The file should be formatted in TOML.

Alternate settings could be stored in other files or constructed at runtime. HTMap provides tools for saving, loading, merging, prepending, and appending settings to each other. Each map is search in order, so earlier settings can flexibly override later settings.

Warning: To entirely replace your settings, do **not** do `htmap.settings = <new settings object>`. Instead, use the *htmap.settings.Settings.replace()* method. Replacing the settings by assignment breaks the internal linking between the settings objects and its dependencies.

Hint: These may be helpful when constructing fresh settings:

- HTMap's base settings are available as `htmap.BASE_SETTINGS`.
- The settings loaded from `~/.htmaprc` are available as `htmap.USER_SETTINGS`.

class htmap.settings.**Settings**(**settings*)

get(*key*, *default=None*)

Return type
Any

to_dict()

Return a single dictionary with all of the settings in this *Settings*, merged according to the lookup rules.

Return type
dict

replace(*other*)

Change the settings of this *Settings* to be the settings from another *Settings*.

Return type
None

append(*other*)

Add a map to the end of the search (i.e., it will be searched last, and be overridden by anything before it).

Parameters
other (Union[*Settings*, dict]) – Another settings-like object to insert into the *Settings*.

Return type
None

prepend(*other*)

Add a map to the beginning of the search (i.e., it will be searched first, and override anything after it).

Parameters
other (Union[*Settings*, dict]) – Another settings-like object to insert into the *Settings*.

Return type
None

classmethod from_settings(**settings*)

Construct a new *Settings* from another *Settings*.

Return type
Settings

classmethod load(*path*)

Load a *Settings* from a file at the given path.

Return type
Settings

save(*path*)

Save this *Settings* to a file at the given path.

Return type
None

6.13 Logging

HTMap exposes a standard Python logging hierarchy under the logger named `'htmap'`. Logging configuration can be done by any of the methods described in the documentation.

Here's an example of how to set up basic console logging:

```
import logging
import sys

logger = logging.getLogger("htmap")
logger.setLevel(logging.DEBUG)

handler = logging.StreamHandler(stream=sys.stdout)
handler.setLevel(logging.DEBUG)
handler.setFormatter(
    logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")
)

logger.addHandler(handler)
```

After executing this code, you should be able to see HTMap log messages as you tell it to do things.

> **Warning:** The HTMap logger is not available in the context of the executing map function. Trying to use it will probably raise exceptions.

6.14 Exceptions

exception htmap.exceptions.**HTMapException**

Base exception for all `htmap` exceptions.

exception htmap.exceptions.**TimeoutError**

An operation has timed out because it took too long.

exception htmap.exceptions.**MissingSetting**

The requested setting has not been set.

exception htmap.exceptions.**OutputNotFound**

The requested output file does not exist.

exception htmap.exceptions.**NoMapYet**

The *htmap.MapBuilder* does not have an associated *htmap.Map* yet because it is still inside the `with` block.

exception htmap.exceptions.**TagAlreadyExists**

The requested `tag` already exists (recover the `Map`, then either use or delete it).

exception htmap.exceptions.**InvalidTag**

The `tag` has an invalid character in it.

exception htmap.exceptions.**TagNotFound**

The requested `tag` does not exist.

exception htmap.exceptions.**EmptyMap**

The map contains no inputs, so it wasn't created.

exception htmap.exceptions.**ReservedOptionKeyword**

The map option keyword you tried to use is reserved by HTMap for internal use.

exception htmap.exceptions.**MisalignedInputData**

There is some kind of mismatch between the lengths of the function arguments and the variadic map options.

exception htmap.exceptions.**CannotRetagMap**

The map cannot be renamed right now.

exception htmap.exceptions.**UnknownPythonDeliveryMethod**

The specified Python delivery method has not been registered.

exception htmap.exceptions.**MapWasRemoved**

This map has been removed, and can no longer be interacted with.

exception htmap.exceptions.**InvalidOutputStatus**

The output status of the map component was not recognized.

exception htmap.exceptions.**MapComponentError**

A map component experienced an error during remote execution.

exception htmap.exceptions.**MapComponentHeld**

A map component has been held by HTCondor.

exception htmap.exceptions.**ExpectedError**

A map component that contained an OK result was unpacked as if it contained an error.

exception htmap.exceptions.**CannotTransplantPython**

The Python interpreter you are using cannot be transplanted.

exception htmap.exceptions.**CannotRerunComponents**

The given components cannot be rerun because they are currently active.

exception htmap.exceptions.**InsufficientHTCondorVersion**

The version of HTCondor is too low to use a feature.

exception htmap.exceptions.**CorruptEventLog**

HTMap doesn't understand what it's seeing in an event log.

6.15 Version

htmap.**version**()

Return a string containing human-readable version information.

Return type
str

htmap.**version_info**()

Return a tuple of version information: `(major, minor, micro, prerelease)`.

Return type
Tuple[int, int, int, Optional[str], Optional[int]]

CHAPTER

SEVEN

CLI REFERENCE

HTMap provides a command line tool called `htmap` that exposes a subset of functionality focused around monitoring long-running maps without needing to run Python yourself.

View the available sub-commands by running:

```
htmap --help  # View available commands
```

Some useful commands are highlighted in the Tips and Tricks section at *Separate Job Submission/Monitoring/Collection*.

Here's the full documentation on all of the available commands:

7.1 htmap

HTMap command line tools.

```
htmap [OPTIONS] COMMAND [ARGS]...
```

Options

-v, --verbose

Show log messages as the CLI runs.

--version

Show the version and exit.

7.1.1 autocompletion

Enable autocompletion for HTMap CLI commands and tags in your shell.

This command should only need to be run once.

Note that your Python environment must be available (i.e., running "htmap" must work) by the time the autocompletion-enabling command runs in your shell configuration file.

```
htmap autocompletion [OPTIONS]
```

Options

--shell <shell>

Required Which shell to enable autocompletion for.

Options

bash | zsh | fish

--force

Append the autocompletion activation command even if it already exists.

--destination <destination>

Append the autocompletion activation command to this file instead of the shell default.

7.1.2 clean

Clean up transient maps by removing them.

Maps that have never had a tag explicitly set are assigned randomized tags and marked as "transient". This command removes maps marked transient (and can also remove all maps, not just transient ones, if the –all option is passed).

```
htmap clean [OPTIONS]
```

Options

--all

Remove non-transient maps as well.

7.1.3 components

Print out the status of the individual components of a map.

```
htmap components [OPTIONS] TAG
```

Options

--status <status>

Print out only components that have this status. Case-insensitive. If not passed, print out the stats of all components (the default).

Options

UNKNOWN | UNMATERIALIZED | IDLE | RUNNING | REMOVED | COMPLETED | HELD | SUSPENDED | ERRORED

--color, --no-color

Toggle colorized output (defaults to colorized).

Arguments

TAG

Required argument

7.1.4 edit

Edit a map's attributes (e.g., its memory request).

Edits do not affect components that are currently running. To "restart" components so that they see the new attribute value, consider vacating their map (see the vacate command).

```
htmap edit [OPTIONS] COMMAND [ARGS]...
```

disk

Set a map's requested disk.

Edits do not affect components that are currently running. To "restart" components so that they see the new attribute value, consider vacating their map (see the vacate command).

```
htmap edit disk [OPTIONS] TAG DISK
```

Options

--unit <unit>

Options
KB | MB | GB

Arguments

TAG

Required argument

DISK

Required argument

memory

Set a map's requested memory.

Edits do not affect components that are currently running. To "restart" components so that they see the new attribute value, consider vacating their map (see the vacate command).

```
htmap edit memory [OPTIONS] TAG MEMORY
```

Options

--unit <unit>

Options
MB | GB

Arguments

TAG

Required argument

MEMORY

Required argument

7.1.5 errors

Show execution error reports for map components.

```
htmap errors [OPTIONS] [TAGS]...
```

Options

-p, --pattern <pattern>
: Act on maps whose tags match glob-style patterns. Pass -p multiple times for multiple patterns.

--all
: Act on all maps.

--limit <limit>
: The maximum number of error reports to show (0, the default, for no limit).

Arguments

TAGS
: Optional argument(s)

7.1.6 hold

This command holds a map. The components of the map will not be allowed to run until released (see the release command).

HTCondor may itself hold your map components if it detects that something has gone wrong with them. Resolve the underlying problem, then use the release command to allow the components to run again.

```
htmap hold [OPTIONS] [TAGS]...
```

Options

-p, --pattern <pattern>
: Act on maps whose tags match glob-style patterns. Pass -p multiple times for multiple patterns.

--all
: Act on all maps.

Arguments

TAGS
Optional argument(s)

7.1.7 logs

Print the path to HTMap's current log file.

The log file rotates, so if you need to go further back in time, look at the rotated log files (stored next to the current log file).

```
htmap logs [OPTIONS]
```

Options

--view, --no-view
If enabled, display the contents of the current log file instead of its path (defaults to disabled).

7.1.8 path

Get paths to parts of HTMap's data storage for a map.

This command is mostly useful for debugging or interfacing with other tools. The tag argument is a map tag, optionally followed by a colon (:) and a target.

If you have a map tagged "foo", these commands would give the following paths (command -> path):

htmap path foo -> the path to the map directory
htmap path foo:map -> also the path to the map directory
htmap path foo:tag -> the path to the map's tag file
htmap path foo:events -> the map's event log
htmap path foo:logs -> directory containing component stdout and stderr
htmap path foo:inputs -> directory containing component inputs
htmap path foo:outputs -> directory containing component outputs

```
htmap path [OPTIONS] TAG
```

Arguments

TAG

Required argument

7.1.9 pause

This command pauses a map. The running components of a paused map will keep their resource claims, but will stop actively executing. The map can be un-paused by resuming it (see the resume command).

```
htmap pause [OPTIONS] [TAGS]...
```

Options

-p, --pattern <pattern>

Act on maps whose tags match glob-style patterns. Pass -p multiple times for multiple patterns.

--all

Act on all maps.

Arguments

TAGS

Optional argument(s)

7.1.10 reasons

Print the hold reasons for map components.

HTCondor may hold your map components if it detects that something has gone wrong with them. Resolve the underlying problem, then use the release command to allow the components to run again.

```
htmap reasons [OPTIONS] [TAGS]...
```

Options

-p, --pattern <pattern>

Act on maps whose tags match glob-style patterns. Pass -p multiple times for multiple patterns.

--all

Act on all maps.

Arguments

TAGS

Optional argument(s)

7.1.11 release

This command releases a map, undoing holds. The held components of a released map will become idle again.

HTCondor may itself hold your map components if it detects that something has gone wrong with them. Resolve the underlying problem, then use this command to allow the components to run again.

```
htmap release [OPTIONS] [TAGS]...
```

Options

-p, --pattern <pattern>

Act on maps whose tags match glob-style patterns. Pass -p multiple times for multiple patterns.

--all

Act on all maps.

Arguments

TAGS

Optional argument(s)

7.1.12 remove

This command removes a map from the Condor queue. Functionally, this command aborts a job.

This function will completely remove a map from the Condor queue regardless of job state (running, executing, waiting, etc). All data associated with a removed map is permanently deleted.

```
htmap remove [OPTIONS] [TAGS]...
```

Options

-p, --pattern <pattern>
: Act on maps whose tags match glob-style patterns. Pass -p multiple times for multiple patterns.

--all
: Act on all maps.

--force
: Do not wait for HTCondor to remove the map components before removing local data.

Arguments

TAGS
: Optional argument(s)

7.1.13 rerun

Rerun (part of) a map from scratch.

The selected components must be completed or errored. See the subcommands of this command group for different ways to specify which components to rerun.

Any existing output of rerun components is removed; they are re-submitted to the HTCondor queue with their original map options (i.e., without any subsequent edits).

```
htmap rerun [OPTIONS] COMMAND [ARGS]...
```

components

Rerun selected components from a single map.

Any existing output of re-run components is removed; they are re-submitted to the HTCondor queue with their original map options (i.e., without any subsequent edits).

```
htmap rerun components [OPTIONS] TAG [COMPONENTS]...
```

Arguments

TAG
: Required argument

COMPONENTS
: Optional argument(s)

map

Rerun all of the components of any number of maps.

Any existing output of re-run components is removed; they are re-submitted to the HTCondor queue with their original map options (i.e., without any subsequent edits).

```
htmap rerun map [OPTIONS] [TAGS]...
```

Options

-p, --pattern <pattern>

Act on maps whose tags match glob-style patterns. Pass -p multiple times for multiple patterns.

--all

Act on all maps.

Arguments

TAGS

Optional argument(s)

7.1.14 resume

This command resumes a map (reverses the pause command). The running components of a resumed map will resume execution on their claimed resources.

```
htmap resume [OPTIONS] [TAGS]...
```

Options

-p, --pattern <pattern>

Act on maps whose tags match glob-style patterns. Pass -p multiple times for multiple patterns.

--all

Act on all maps.

Arguments

TAGS
Optional argument(s)

7.1.15 retag

Change the tag of an existing map.

Retagging a map makes it not transient. Maps that have never had an explicit tag given to them are transient and can be easily cleaned up via the clean command.

```
htmap retag [OPTIONS] TAG NEW
```

Arguments

TAG
Required argument

NEW
Required argument

7.1.16 set

Change a setting in your ~/.htmaprc file.

```
htmap set [OPTIONS] SETTING VALUE
```

Arguments

SETTING
Required argument

VALUE
Required argument

7.1.17 settings

Print HTMap's current settings.

By default, this command shows the merger of your user settings from ~/.htmaprc and HTMap's own default settings. To show only your user settings, pass the –user option.

```
htmap settings [OPTIONS]
```

Options

--user
Display only user settings (the contents of ~/.htmaprc).

7.1.18 status

Print a status table for all of your maps.

Transient maps are prefixed with a leading "*".

```
htmap status [OPTIONS]
```

Options

--state, --no-state
Toggle display of component states (defaults to enabled).

--meta, --no-meta
Toggle display of map metadata like memory, runtime, etc. (defaults to enabled).

--format <format>
Select output format: plain text, JSON, compact JSON, or CSV (defaults to plain text)

Options
text | json | json_compact | csv

--live, --no-live
Toggle live reloading of the status table (defaults to not live).

--color, --no-color
Toggle colorized output (defaults to colorized).

7.1.19 stderr

Look at the stderr for a map component.

```
htmap stderr [OPTIONS] TAG COMPONENT
```

Options

--timeout <timeout>
How long to wait (in seconds) for the file to be available. If not set (the default), wait forever.

Arguments

TAG
Required argument

COMPONENT
Required argument

7.1.20 stdout

Look at the stdout for a map component.

```
htmap stdout [OPTIONS] TAG COMPONENT
```

Options

--timeout <timeout>
How long to wait (in seconds) for the file to be available. If not set (the default), wait forever.

Arguments

TAG
Required argument

COMPONENT
Required argument

7.1.21 tags

Print the tags of existing maps.

```
htmap tags [OPTIONS]
```

Options

-p, --pattern <pattern>

Act on maps whose tags match glob-style patterns. Patterns must be enclosed in “”. Pass -p multiple times for multiple patterns.

7.1.22 transplants

Manage transplant installs.

```
htmap transplants [OPTIONS] COMMAND [ARGS]...
```

info

Display information on available transplant installs.

```
htmap transplants info [OPTIONS]
```

remove

Remove a transplant install by index.

```
htmap transplants remove [OPTIONS] INDEX
```

Arguments

INDEX

Required argument

7.1.23 vacate

This command vacates a map. The running components of a vacated map will give up their claimed resources and become idle again.

Checkpointing maps will still have access to their last checkpoint, and will resume from it as if execution was interrupted for any other reason.

```
htmap vacate [OPTIONS] [TAGS]...
```

Options

-p, --pattern <pattern>
Act on maps whose tags match glob-style patterns. Pass -p multiple times for multiple patterns.

--all
Act on all maps.

Arguments

TAGS
Optional argument(s)

7.1.24 version

Print HTMap and HTCondor Python bindings version information.

```
htmap version [OPTIONS]
```

7.1.25 wait

Wait for maps to complete.

```
htmap wait [OPTIONS] [TAGS]...
```

Options

-p, --pattern <pattern>
Act on maps whose tags match glob-style patterns. Pass -p multiple times for multiple patterns.

--all
Act on all maps.

Arguments

TAGS
Optional argument(s)

CHAPTER

EIGHT

SETTINGS

HTMap's settings are controlled by a global object which you can access as `htmap.settings`. For more information on how this works, see *htmap.settings.Settings*.

Users can provide custom default settings by putting them in a file in their home directory named `.htmaprc`. The file is in TOML format.

HTMap can also read certain settings from the environment. When this is possible, it is noted in the description of the setting.

The precedence order is that runtime settings override `.htmaprc` settings, which override environment settings, which override built-in defaults.

HTMap's settings are organized into groupings based on TOML headers. The settings inside each group are discussed in the following sections.

At runtime, settings can be found via dotted paths that correspond to the section heads. Here, I'll give the dotted paths - if they're in the file instead, each dot is a header.

Here is an example `.htmaprc` file:

```
DELIVERY_METHOD = "docker"

[MAP_OPTIONS]
REQUEST_MEMORY = "250MB"

[DOCKER]
IMAGE = "python:latest"
```

The equivalent runtime Python commands to set those settings would be

```
import htmap

htmap.settings["DELIVERY_METHOD"] = "docker"
htmap.settings["MAP_OPTIONS.REQUEST_MEMORY"] = "250MB"
htmap.settings["DOCKER.IMAGE"] = "python:latest"
```

8.1 Settings

These are the top-level settings. They do not belong to any header.

`HTMAP_DIR` - the path to the directory to use as the HTMap directory. If not given, defaults to `~/.htmap`.

`DELIVERY_METHOD` - the name of the delivery method to use. The different delivery methods are discussed in *Dependency Management*. Defaults to `docker`. Inherits the environment variable `HTMAP_DELIVERY`.

`WAIT_TIME` - how long to wait between polling for component statuses, files existing, etc. Measured in seconds. Defaults to `1` (1 second).

`CLI` - set to `True` automatically when HTMap is being used from the CLI. Defaults to `False`.

8.1.1 MAP_OPTIONS

Any settings in this section are passed to every `MapOption` as keyword arguments.

8.1.2 HTCONDOR

`SCHEDULER` - the address of the HTCondor scheduler (see `htcondor.Schedd`). If set to `None`, HTMap discovers the local scheduler automatically. Defaults to `None`. Inherits the environment variable `HTMAP_CONDOR_SCHEDULER`.

`COLLECTOR` - the address of the HTCondor collector (see `htcondor.Collector`). If set to `None`, HTMap discovers the local collector automatically. Defaults to `None`. Inherits the environment variable `HTMAP_CONDOR_COLLECTOR`.

8.1.3 DOCKER

These settings control how the `docker` delivery method works.

`IMAGE` - the path to the Docker image to run components with. Defaults to `'continuumio/anaconda3:latest'`. If the environment variable `HTMAP_DOCKER_IMAGE` is set, that will be used as the default instead.

8.1.4 SINGULARITY

These settings control how the `singularity` delivery method works.

`IMAGE` - the path to the Singularity image to run components with. Defaults to `'docker://continuumio/anaconda3:latest'`. If the environment variable `HTMAP_SINGULARITY_IMAGE` is set, that will be used as the default instead.

8.1.5 TRANSPLANT

These settings control how the `transplant` delivery method works.

`DIR` - the path to the directory where the zipped Python install will be cached. Defaults to a subdirectory of `HTMAP_DIR` named `transplants`.

`ALTERNATE_INPUT_PATH` - a string that will be used in the HTCondor `transfer_input_files` option **instead of** the local file path. If set to `None`, the local path will be used (the default). This can be used to override the default file transfer mechanism.

`ASSUME_EXISTS` - if set to `True`, assume that the zipped Python install already exists. Most likely, you will need to set `ALTERNATE_INPUT_PATH` to an existing zipped install. Defaults to `False`.

CHAPTER

NINE

DEPENDENCY MANAGEMENT

Dependency management for Python programs is a thorny issue in general, and running code on computers that you don't own is even thornier. HTMap provides several methods for ensuring that the software that your code depends on is available for your map components. This could include other Python packages like `numpy` or `tensorflow`, or external software like `gcc`.

There are two halves of the dependency management game. The first is on "your" computer, which we call **submit-side**. This could be your laptop running a personal HTCondor pool, or an HTCondor "submit node" that you `ssh` to, or whatever other way you access your HTCondor pool. The other side is **execute-side**, which isn't really a single place: it is all of the execute nodes in the pool that your map components might run on.

Submit-side dependency management can be handled using standard Python package management tools. We recommend using `miniconda` as your package manager (https://docs.conda.io/en/latest/miniconda.html).

HTMap itself requires that execute-side can run a Python script using a Python install that also has `htmap` installed. That Python installation also needs whatever other packages your code needs to run. For example, if you `import numpy` in your code, you need to have `numpy` installed execute-side.

As mentioned above, HTMap provides several "delivery methods" for getting that Python installation to the execute location. The built-in delivery methods are

- `docker` - runs in a (possibly user-supplied) Docker container.
- `singularity` - runs in a (possibly user-supplied) Singularity container.
- `shared` - runs with the same Python installation used submit-side.
- `assume` - assumes that the dependencies have already been installed at the execute location.
- `transplant` - copy the submit-side Python installation to the execute location.

More details on each of these methods can be found below.

The default delivery method is `docker`, with the default image `htcondor/htmap-exec:<version>`, where version will match the version of HTMap you are using submit-side. If your pool can run Docker jobs and your Python code does not depend on any custom packages (i.e., you never import any modules that you wrote yourself), this default behavior will likely work for you without requiring any changes. See the section below on Docker if this isn't the case!

Attention: HTMap can transfer inputs and outputs between different minor versions of Python 3, but it can't magically make features from later Python versions available. For example, if you run Python 3.6 submit-side you can use f-strings in your code. But if you use Python 3.5 execute-side, your code will hit syntax errors because f-strings were not added until Python 3.6. We don't actually test cross-version transfers though, and we recommend running exactly the same version of Python on submit and execute.

HTMap **cannot** transfer inputs and outputs between different versions of `cloudpickle`. Ensure that you have the same version of `cloudpickle` installed everywhere.

If you see an exception on a component related to `cloudpickle.load`, this is the most likely culprit. Note that you may need to manually upgrade/downgrade your submit-side or execute-side `cloudpickle`.

9.1 Run Inside a Docker Container

In your `~/.htmaprc` file:

```
DELIVERY_METHOD = "docker"

[DOCKER]
IMAGE = "<repository>/<image>:<tag>"
```

At runtime:

```
htmap.settings["DELIVERY_METHOD"] = "docker"
htmap.settings["DOCKER.IMAGE"] = "<repository>/<image>:<tag>"
```

In this mode, HTMap will run inside a Docker image that you provide. Remember that this Docker image needs to have the `htmap` module installed. The default Docker image is htcondor/htmap-exec, which is based on Python 3 and has many useful packages pre-installed.

If you want to use your own Docker image, just change the `'DOCKER.IMAGE'` setting. Your Docker image needs to be pushed back to Docker Hub (or some other Docker image registry that your HTCondor pool can access) to be usable. For example, a very simple Dockerfile that can be used with HTMap is

```
FROM python:3

RUN pip install --no-cache-dir htmap
```

This would create a Docker image with the latest versions of Python 3 and `htmap` installed. From here you could install more Python dependencies, or add more layers to account for other dependencies.

Attention: More information on building Docker images for use with HTMap can be found in the *Docker Image Cookbook*.

Warning: This delivery mechanism will only work if your HTCondor pool supports Docker jobs! If it doesn't, you'll need to talk to your pool administrators or use a different delivery mechanism.

9.2 Run Inside a Singularity Container

In your `~/.htmaprc` file:

```
DELIVERY_METHOD = "singularity"

[SINGULARITY]
IMAGE = "<image>"
```

At runtime:

```
htmap.settings["DELIVERY_METHOD"] = "singularity"
htmap.settings["SINGULARITY.IMAGE"] = "<image>"
```

In this mode, HTMap will run inside a Singularity image that you provide. Remember that this Singularity image needs to have the `cloudpickle` module installed.

Singularity can also use Docker images. Specify a Docker Hub image as `htmap.settings['SINGULARITY.IMAGE'] = "docker://<repository>/<image>:<tag>"` to download a Docker image from DockerHub and automatically use it as a Singularity image.

For consistency with Docker delivery, the default Singularity image is docker://continuumio/anaconda3:latest, which has many useful packages pre-installed.

If you want to use your own Singularity image, just change the `'SINGULARITY.IMAGE'` setting.

Warning: This delivery mechanism will only work if your HTCondor pool supports Singularity jobs! If it doesn't, you'll need to talk to your pool administrators or use a different delivery mechanism.

Note: When using this delivery method, HTMap will discover `python3` on the system `PATH` and use that to run your code.

Warning: This delivery method relies on the directory `/htmap/scratch` either existing in the Singularity image, or Singularity being able to run with `overlayfs`. If you get a `stderr` message from Singularity about a bind mount directory not existing, that's the problem.

9.3 Run With a Shared Python Installation

In your `~/.htmaprc` file:

```
DELIVERY_METHOD = "shared"
```

At runtime:

```
htmap.settings["DELIVERY_METHOD"] = "shared"
```

In this mode, HTMap will run your components using the same interpreter being used submit-side. This requires that that the submit-side Python interpreter be "visible" from the execute location, which is usually done in one of two ways:

1. The execute location **is** the submit location (i.e., they are the same physical computer).
2. The Python installation is stored on a shared filesystem, such that submit and execute can both see the same file paths.

Either way, the practical requirement to use this delivery method is that the path to the Python interpreter (i.e., `python -c "import sys, print(sys.executable)"`) is the same both submit-side and execute-side.

9.4 Assume Dependencies are Present

In your `~/.htmaprc` file:

```
DELIVERY_METHOD = "assume"
```

At runtime:

```
htmap.settings["DELIVERY_METHOD"] = "assume"
```

In this mode, HTMap assumes that a Python installation with all Python dependencies is already present. This will almost surely require some additional setup by your HTCondor pool's administrators.

9.5 Transplant Existing Python Install

In your `~/.htmaprc` file:

```
DELIVERY_METHOD = "transplant"
```

At runtime:

```
htmap.settings["DELIVERY_METHOD"] = "transplant"
```

If you are running HTMap from a standalone Python install (like an Anaconda installation), you can use this delivery mechanism to transfer a copy of your entire Python install. All locally-installed packages (including `pip -e` "editable" installs) will be available.

For advanced transplant functionality, see *TRANSPLANT*.

Note: The first time you run a map after installing/removing packages, you will need to wait while HTMap re-zips your installation. Subsequent maps will use the cached version.

HTMap uses `pip` to check whether the cached Python is current, so make sure that `pip` is installed in your Python.

Warning: This mechanism does not work with system Python installations (which you shouldn't be using anyway!).

Note: When using the transplant method the transplanted Python installation will be used to run the component, regardless of any other Python installations that might exist execute-side.

CHAPTER

TEN

VERSION HISTORY

10.1 v0.6.1

This version is a drop-in replacement for v0.6.0, except that it relaxes the version requirements for several dependencies to accommodate upcoming changes to the pip dependency resolver.

10.1.1 Known Issues

- HTMap does not currently allow "directory content transfers", which is an HTCondor feature where naming a directory in `transfer_input_files` with a trailing slash transfers the contents of the directory, not the directory itself. (If you try it, the directory itself will be transferred, as if you had not used a trailing slash). Issue: #215
- Execution errors that result in the job being terminated but no output being produced are still not handled entirely gracefully. Right now, the component state will just show as `ERRORED`, but there won't be an actual error report.
- Map component state may become corrupted when a map is manually vacated. Force-removal may be needed to clean up maps if HTCondor and HTMap disagree about the state of their components. Issue: #129

10.2 v0.6.0

The big new features in this release are:

- Improved support for input and output file transfer (inputs/outputs can come from/be sent to remote locations, i.e., not the submit machine).
- A new delivery method, `shared`, where HTMap will use the same Python executable detected submit-side when executing (this supports HTCondor pools that use shared filesystems to make a Python installation universally available).

10.2.1 New Features/Improvements

- Add the `shared` delivery method, which supports HTCondor pools that use shared filesystems to make Python installations available universally. Suggested by Duncan Macleod. Issues/PRs: #195, #198, #200
- HTMap now supports getting input files from remote destinations (i.e., not from the submit machine) via existing input file auto-discovery. Just use the revised *`TransferPath`* in your mapped function arguments, and HTMap will arrange for the file to be transferred to your map component! PR: #216
- HTMap now supports sending output files to destinations that are not the submit machine via HTCondor's `transfer_output_remaps` mechanism. Output files can be sent to various locations, such as an S3 service. See the new `output_remaps` argument of *`MapOptions`* and the revised *`TransferPath`*, as well as the new tutorial *Transferring Output to Other Places* for more details on how to use this feature. PR: #216
- **Massive** documentation upgrades courtesy of Scott Sievert! Issues/PRs: #208, #191, #202, #221
- The HTMap CLI (normally accessed by running `htmap`) can now also be accessed by running `python -m htmap`. Issue: #190
- The HTMap CLI now supports autocompletion on commands and tags. Run `htmap autocompletion` from the command line to add the necessary setup to your shell startup script.
- The HTMap CLI `logs` command now has a `--view` option which, instead of just printing the path to the HTMap log file, displays its contents.

10.2.2 Changed/Deprecated Features

- *`htmap.Map.exists`* has replaced `htmap.Map.is_removed`. It has exactly the opposite semantics (it is only `True` if the map has not been successfully removed). PR: #221
- *`htmap.ComponentStatus`* is now a subclass of `str`, so (for example) `"COMPLETED"` can be used in place of `htmap.ComponentStatus.COMPLETED`.
- Item access (`[]`) on `Map.stdout`, `Map.stderr`, and `Map.output_files` is now non-blocking and will raise `FileNotFound` exceptions if accessed before available. The blocking API (with a timeout) is still available via the `get` method.
- The HTMap CLI `version` command now also prints HTCondor Python bindings version information. Added `htmap --version` that only prints HTMap version information.
- Several HTMap CLI commands now support explicit enable/disable flags instead of just one or the other. The default behaviors were not changed.
- The name of the function used to register delivery methods changed to *`register_delivery_method()`* (from `register_delivery_mechanism`).

10.2.3 Bug Fixes

- HTMap is now less sensitive to job event logs becoming corrupted.
- Type hints are now more correct on more functions (but not fully correct on all functions, bear with us!).

10.2.4 Known Issues

- HTMap does not currently allow "directory content transfers", which is an HTCondor feature where naming a directory in `transfer_input_files` with a trailing slash transfers the contents of the directory, not the directory itself. (If you try it, the directory itself will be transferred, as if you had not used a trailing slash). Issue: #215
- Execution errors that result in the job being terminated but no output being produced are still not handled entirely gracefully. Right now, the component state will just show as `ERRORED`, but there won't be an actual error report.
- Map component state may become corrupted when a map is manually vacated. Force-removal may be needed to clean up maps if HTCondor and HTMap disagree about the state of their components. Issue: #129

10.3 v0.5.1

10.3.1 New Features

10.3.2 Deprecated Features

10.3.3 Bug Fixes

- Maps can now be force-removed even if the schedd cannot be contacted. Graceful removal still requires contact with the schedd. Issue: https://github.com/htcondor/htmap/issues/186

10.3.4 Known Issues

- Execution errors that result in the job being terminated but no output being produced are still not handled entirely gracefully. Right now, the component state will just show as `ERRORED`, but there won't be an actual error report.
- Map component state may become corrupted when a map is manually vacated. Force-removal may be needed to clean up maps if HTCondor and HTMap disagree about the state of their components. Issue: https://github.com/htcondor/htmap/issues/129

10.4 v0.5.0

10.4.1 New Features

- HTMap CLI commands that operate on tags can now pattern-match for tags using glob syntax. Try adding `-p "<pattern>"` to commands like `htmap remove` or `htmap release`! Issue: https://github.com/htcondor/htmap/issues/159
- Component status tracking is now preserved between sessions, so it won't be performed from scratch every time. This will only work if the HTCondor Python bindings version is 8.9.3 or greater. You can upgrade your bindings version roughly-independently of HTMap by running `pip install --upgrade htcondor`. Issue: https://github.com/htcondor/htmap/issues/166
- *htmap.Map*, *htmap.MapStdOut*, *htmap.MapStdErr*, and *htmap.MapOutputFiles* now all support in the `in` operator to check if a component index is in the map.

10.4.2 Deprecated Features

- The various iteration methods on *htmap.Map* no longer have a `callback` argument.

10.4.3 Bug Fixes

- It should now be much harder to accidentally get a dangling, inaccessible map due to an interrupted `remove`. Issue: https://github.com/htcondor/htmap/issues/127
- When an execution errors occurs, the exception and traceback will be printed to stderr execute-side (in addition to being brought back submit-side). This should make some debugging patterns work as expected. Issue: https://github.com/htcondor/htmap/issues/178
- The CLI command `htmap status --live` now has much better behavior when the table width is nearly the width of the terminal. It should now never wrap unless the table is actually wider than the terminal, instead of a few characters before the actual width.
- HTMap now handles late materialized jobs much more smoothly: maps with unmaterialized components can be removed, and various CLI commands that output color won't fail when acting on maps with unmaterialized components. However, unmaterialized components do not show as *IDLE*, which mirrors the behavior of *condor_q*. This does make it hard to know how many components are in a late-materialized map at a glance; we are thinking about how to address this. Issue: https://github.com/htcondor/htmap/issues/158

10.4.4 Known Issues

- Execution errors that result in the job being terminated but no output being produced are still not handled entirely gracefully. Right now, the component state will just show as `ERRORED`, but there won't be an actual error report.
- Map component state may become corrupted when a map is manually vacated. Force-removal may be needed to clean up maps if HTCondor and HTMap disagree about the state of their components. Issue: https://github.com/htcondor/htmap/issues/129

10.5 v0.4.4

10.5.1 New Features

10.5.2 Bug Fixes

- In execution error reports, local variables with very long string forms are now cut down to a smaller size.

10.5.3 Known Issues

- Execution errors that result in the job being terminated but no output being produced are still not handled entirely gracefully. Right now, the component state will just show as `ERRORED`, but there won't be an actual error report.
- Map component state may become corrupted when a map is manually vacated. Force-removal may be needed to clean up maps if HTCondor and HTMap disagree about the state of their components. Issue: https://github.com/htcondor/htmap/issues/129

10.6 v0.4.3

10.6.1 New Features

10.6.2 Bug Fixes

- CLI *stdout* and *stderr* commands were broken, but are now fixed.
- Add the missing parts of the */.singularity.d* directory that will make *v0.4.2* Singularity support actually work.

10.6.3 Known Issues

- Execution errors that result in the job being terminated but no output being produced are still not handled entirely gracefully. Right now, the component state will just show as `ERRORED`, but there won't be an actual error report.
- Map component state may become corrupted when a map is manually vacated. Force-removal may be needed to clean up maps if HTCondor and HTMap disagree about the state of their components. Issue: https://github.com/htcondor/htmap/issues/129

10.7 v0.4.2

10.7.1 New Features

10.7.2 Bug Fixes

- *Map.errors* and *Map.error_reports()* now work when there is a mix of holds and errors in the map. Previously, held components would cause both of these to raise *MapComponentHeld* when trying to access them in that situation. Issue: https://github.com/htcondor/htmap/issues/165
- Requirements statement merging was broken when any of the three sources of requirements (settings, function-level map options, and individual-map map options) were not given. Requirements from all source are now properly merged, regardless of whether any of them actually exist. Issue: https://github.com/htcondor/htmap/issues/168
- Top-level settings that were dictionaries (like `MAP_OPTIONS`) did not behave correctly when elements of them were set; they did not inherit the old settings. These kinds of settings are now properly inherited, but expect breaking changes in the *Settings* API next release to resolve the underlying issues. Issue: https://github.com/htcondor/htmap/issues/169
- The `htmap-exec` Docker image should now cleanly export to Singularity. Issue: https://github.com/htcondor/htmap/issues/173

10.7.3 Known Issues

- Execution errors that result in the job being terminated but no output being produced are still not handled entirely gracefully. Right now, the component state will just show as `ERRORED`, but there won't be an actual error report.
- Map component state may become corrupted when a map is manually vacated. Force-removal may be needed to clean up maps if HTCondor and HTMap disagree about the state of their components. Issue: https://github.com/htcondor/htmap/issues/129

10.8 v0.4.1

10.8.1 New Features

10.8.2 Bug Fixes

- Fixed a bug where maps submitted with late materialization would choke on the "cluster submit" event when reading their event log. Band-aided for now.

10.8.3 Known Issues

- Execution errors that result in the job being terminated but no output being produced are still not handled entirely gracefully. Right now, the component state will just show as `ERRORED`, but there won't be an actual error report.
- Map component state may become corrupted when a map is manually vacated. Force-removal may be needed to clean up maps if HTCondor and HTMap disagree about the state of their components. Issue: https://github.com/htcondor/htmap/issues/129

10.9 v0.4.0

10.9.1 New Features

- **HTMap can now transfer output files!** See the new recipe: *Output Files* and the new *htmap.transfer_output_files()* function.
- HTMap's default Docker image is now `htcondor/htmap-exec`, which is produced from a Dockerfile in the HTMap `git` repository. It is based on `continuumio/anaconda3`, with `htmap` itself installed as well. Issue: https://github.com/htcondor/htmap/issues/152
- Redid *htmap.Map* `stdout` and `stderr`. They are now attributes that represent sequences over the `stdout` and `stderr` from the map components, as strings, respectively.
- Acts and Edits on Maps that are not "active" (i.e., have components in the HTCondor queue) are now no-ops. Includes a new `htmap.Map.is_active` property, which is `True` if any components are still in the queue. Issue: https://github.com/htcondor/htmap/issues/145

10.9.2 Bug Fixes

10.9.3 Known Issues

- Execution errors that result in the job being terminated but no output being produced are still not handled entirely gracefully. Right now, the component state will just show as `ERRORED`, but there won't be an actual error report.

- Map component state may become corrupted when a map is manually vacated. Force-removal may be needed to clean up maps if HTCondor and HTMap disagree about the state of their components. Issue: https://github.com/htcondor/htmap/issues/129

10.10 v0.3.2

10.10.1 New Features

10.10.2 Bug Fixes

- Hopefully finally resolved a recurring issue with checkpoint directories being returned to the submit node after execution errors. Issue: https://github.com/htcondor/htmap/issues/128
- *htmap.Map.error_reports()* can now get error reports while part of a map is still running.

10.10.3 Known Issues

- Execution errors that result in the job being terminated but no output being produced are still not handled entirely gracefully. Right now, the component state will just show as `ERRORED`, but there won't be an actual error report.
- Map component state may become corrupted when a map is manually vacated. Force-removal may be needed to clean up maps if HTCondor and HTMap disagree about the state of their components. Issue: https://github.com/htcondor/htmap/issues/129

10.11 v0.3.1

10.11.1 New Features

10.11.2 Bug Fixes

- Live status display will no longer explode if you remove a map out from under it. Issue: https://github.com/htcondor/htmap/issues/144
- Fix new `htmap rerun` command.

10.11.3 Known Issues

- Execution errors that result in the job being terminated but no output being produced are still not handled entirely gracefully. Right now, the component state will just show as `ERRORED`, but there won't be an actual error report.
- Map component state may become corrupted when a map is manually vacated. Force-removal may be needed to clean up maps if HTCondor and HTMap disagree about the state of their components. Issue: https://github.com/htcondor/htmap/issues/129

10.12 v0.3.0

10.12.1 New Features

- Revised internals on how error information is returned from execute nodes. HTMap now detects runtime errors during component status checks (without too much overhead).
- Add `singularity` delivery method. More revisions needed to use best practices, but it works. Expect major changes in the future…
- Add `htmap components` CLI command, which can print out individual component statuses for a map. For example, `htmap components <tag>` will print out all of the components for a map and their statuses. `htmap components --status ERRORED <tag>` will print out only the components whose status is ERRORED.
- Some execution errors (especially the kind that result in output not being produced) are now turned into holds by using the submit descriptor `ON_EXIT_HOLD`.
- Reworked CLI `rerun` command. It now has separate sub-commands for rerunning entire maps or only certain components.

10.12.2 Bug Fixes

10.12.3 Known Issues

- Execution errors that result in the job being terminated but no output being produced are still not handled entirely gracefully. Right now, the component state will just show as `ERRORED`, but there won't be an actual error report.
- Map component state may become corrupted when a map is manually vacated. Force-removal may be needed to clean up maps if HTCondor and HTMap disagree about the state of their components. Issue: https://github.com/htcondor/htmap/issues/129

CHAPTER

ELEVEN

CONTRIBUTING AND DEVELOPING

HTMap is open to contributions! Please feel free to submit a Pull Request on GitHub. If you contribute to HTMap, please add your name to the `CONTRIBUTORS` file in the repository root (if you want to be listed).

Development Environment
: How to set up an environment for development and testing.

HTMap Innards
: How HTMap does what it does.

How to Release a New HTMap Version
: How to release a new version of HTMap.

11.1 HTMap Innards

11.1.1 Overview

HTMap turns Python functions into HTCondor jobs. There are two levels of wrapping that it does: the function call, with its inputs and outputs (including file transfer) and any possible errors, are implicitly wrapped, but most other HTCondor features, like resource requests and custom submit descriptors, are presented more directly (though still with a Python-oriented interface).

The distinction between these two levels was chosen to provide the maximum amount of "do the expected thing" with the Python parts of running the job while allowing maximum flexibility for the HTCondor parts of the running job. There is no hard line, and different parts of this have moved back and forth over the line during development (namely file transfer, but at one point resource requests were also treated specially).

11.1.2 Guiding Principles

- The only identifying piece of information about a map a **user** should ever need is a `tag`.
- Users should never have to directly interact with the filesystem to look at any information about their map.
- We should store as little state as possible in memory. Recalculating state of anything but the very largest maps is very fast.

- Any state we do store should be duplicated on disk immediately. It should be possible to resubmit (any part of) a map based only on information stored on disk.

11.1.3 Moving Things Around

HTMap relies on cloudpickle to move data back and forth the submit node and execute nodes. It pickles the Python function that the user provides as well as all of the input, then turns around and submits an HTCondor job cluster using HTCondor's Python bindings. Instead of directly running user scripts, HTMap uses a script that it controls as the HTCondor executable. It hands the user back an object that can be used to look at the output of the function as well as control the execution of the underlying cluster jobs.

11.1.4 The `run` Subdirectory

For basic functionality, HTMap itself does not need to be installed on the execute node where jobs it creates run. This offers the advantage of being using to use Docker images that only contain `cloudpickle` (which is many, because it's installed as part of the Anaconda distribution) without modification. Currently, if you want to use checkpointing or output file transfer, you must also install HTMap execute-side. In practice, we expect people to install HTMap in their execute image, and all of the instructions in the docs say to do so.

To accomplish this decoupling, HTMap uses a Python script as its HTCondor executable that has no dependencies except the Python standard library and `cloudpickle`. This script is stored inside the library at `htmap/run/run.py`. The transplant delivery method wraps this script with `htmap/run/run_with_transplant.sh`, a `bash` script that handles unpacking the transplanted install. A similar script exists for Singularity.

It is critical that the `run.py` script make all possible efforts to exit without an error. If the script itself generates an error, it tends to become very difficult for users to understand what went wrong. For example, we used to `import cloudpickle` in the bag of imports at the top of the script. If `cloudpickle` wasn't present in the execute image, the script would immediately bail out and HTMap wouldn't understand why; the user would have to inspect the `stderr` of the map component (which also wasn't directly supported at the time) to understand what went wrong.

11.1.5 Data Model

Each **map** is tied to a **map directory**, which is named by a UUID. The map directories are stored in a subdirectory of the **HTMap directory**. The HTMap directory is located according to `settings['HTMAP_DIR']` (default `~/.htmap`).

The human-readable name of each map is its **tag**. Tags are stored in a different subdirectory of the HTMap directory, which acts a file-based map between tags and the names of the map directories. Each tag file's name is that map's tag, and the file's contents are the name of the map directory.

All input, output, and HTCondor metadata (event logs, for example) for a map is stored in its map directory. A single input/output pair is a **component**, and the components of a map are just referred to by their index in the input iterable.

11.1.6 Serializing and Deserializing Data

HTMap uses a wide variety of data serialization formats, depending on what needs to be stored. The names of the directories and files can be found in `htmap/names.py`. They are all stored inside the map's directory.

The **itemdata** for each map is stored as a JSON-formatted list of dictionaries. The itemdata is used to call `htcondor.Submit.queue_with_itemdata()` during map creation.

The **submit object** for each map is stored as a JSON-formatted dictionary.

The **number of components** is stored as a single string-ified integer in the file.

The **cluster IDs** of each HTCondor cluster job associated with the map are stored as newline-separated plain-text strings.

The **event log** for each HTCondor cluster job is routed to a file inside the map directory.

For generic data, like the **inputs** and **outputs** of mapped functions, HTMap uses `cloudpickle`. The individual inputs and outputs for each component are stored in files named by the component index.

The functions that handle storing and loading these various formats are in the `htmap.htio` submodule. All IO should go through methods defined in that submodule, with the idea that if it becomes necessary to swap out some of the internal implementations of those methods, the changes will be isolated to that module.

`htmap.htio.save_object`(*obj*, *path*)

Serialize a Python object (including "objects", like functions) to a file at the given `path`.

Return type
`None`

`htmap.htio.load_object`(*path*)

Deserialize an object from the file at the given `path`.

Return type
`Any`

`htmap.htio.load_objects`(*path*)

Deserialize a stream of objects from the file at the given `path`.

Return type
`Iterator`[`Any`]

`htmap.htio.save_func`(*map_dir*, *func*)

Save the mapped function to the map directory.

Return type
`None`

`htmap.htio.save_inputs`(*map_dir*, *args_and_kwargs*)

Save the arguments to the mapped function to the map's input directory.

Return type
`None`

htmap.htio.**save_num_components**(*map_dir*, *num_components*)

Save the number of components in a map.

Return type
None

htmap.htio.**load_num_components**(*map_dir*)

Load the number of components in a map.

Return type
int

htmap.htio.**append_cluster_id**(*map_dir*, *cluster_id*)

Add a cluster ID to a map.

Return type
None

htmap.htio.**load_cluster_ids**(*map_dir*)

Load the cluster IDs for a map.

Return type
List[int]

htmap.htio.**save_submit**(*map_dir*, *submit*)

Save a dictionary that represents the map's `htcondor.Submit` object.

Return type
None

htmap.htio.**load_submit**(*map_dir*)

Load an `htcondor.Submit` object that was saved using *save_submit()*.

Return type
Submit

htmap.htio.**save_itemdata**(*map_dir*, *itemdata*)

Save the map's itemdata as a list of JSON dictionaries.

Return type
None

htmap.htio.**load_itemdata**(*map_dir*)

Load itemdata that was saved using *save_itemdata()*.

Return type
List[dict]

11.2 Development Environment

11.2.1 Repository Setup

You can get HTMap's source code by cloning the git repository: `git clone https://github.com/htcondor/htmap`. If you are planning on submitting a pull request, you should instead clone your own fork of the repository.

After cloning the repository, install the development dependencies using your Python package manager. If you are using `pip`, you would run `pip install -e .[tests,docs]` from the repository root. The dependencies (development and otherwise) are listed in `setup.cfg`.

> **Warning:** The HTCondor Python bindings are currently only available via PyPI on Linux. On Windows you must install HTCondor itself to get them. On Mac, you're out of luck. Install `pre-commit` manually, then use the development container to run the test suite/build the documentation.

One of the dependencies you just installed is `pre-commit`. `pre-commit` runs a series of checks whenever you try to commit. You should "install" the pre-commit hooks by running `pre-commit install` in the repository root. You can run the checks manually at any time by running `pre-commit`.

Do not commit to the repository before running `pre-commit install`!

11.2.2 Development Container

HTMap's test suite relies on a properly set-up environment. The simplest way to get that environment is to use the Dockerfile in `docker/Dockerfile` to produce a **development container**. The repository includes a bash script named `dr` (**d**ocker **r**un) in the repository root that will let you quickly build and execute commands in a development container.

> **Attention:** The `dr` script bind-mounts your local copy of the repository into the container. Any edits you make outside the container will be reflected inside (and vice versa).

Anything you pass to `dr` will be executed inside the container. By default (i.e., if you pass nothing) you will get a `bash` shell. The initial working directory is the `htmap` repository inside the container.

11.2.3 Running the Test Suite

The best way to run the test suite is to run `pytest` inside the development container:

```
$ ./dr
# ...
mapper@161b6af91d72:~/htmap$ pytest
```

The test suite can be executed in parallel by passing the `-n` option. `pytest -n 4` seems to be a good number for laptops, while desktops can probably handle `-n 10`. See pytest-xdist for more details on parallel execution. The test suite is very slow when run serially; we highly recommend running with a large number of workers.

See the pytest docs or run `pytest --help` for more information on *pytest* itself.

11.2.4 Building the Docs

HTMap's documentation is served by Read the Docs, which builds the docs as well. The docs are deployed automatically on each commit to master, so they can be updated independently of a version release for minor adjustments.

It can be helpful to build the docs locally during development. We recommend using `sphinx-autobuild` to serve the documentation via a local webserver and automatically rebuild the documentation when changes are made to the package source code or the documentation itself. To run the small wrapper script we have written around `sphinx-autobuild`, from inside or outside the development container run,

```
$ ./dr
# ...
mapper@161b6af91d72:~/htmap$ docs/autobuild.sh
NOTE: CONNECT TO http://127.0.0.1:8000 NOT WHAT SPHINX-AUTOBUILD TELLS YOU
# trimmed; visit URL above
```

Note the startup message: ignore the link that `sphinx-autobuild` gives you, and instead go to http://127.0.0.1:8000 to see the built documentation.

11.2.5 Binder Integration

HTMap's tutorials can be served via Binder. The tutorials are run inside a specialized Docker container (not the development container). To test whether the Binder container is working properly, run the `binder/run.sh` script from the repository root (i.e., not from inside the development container):

```
$ ./binder/run.sh
```

It will give you a link to enter into your web browser that will land you in the same Jupyter environment you would get on Binder.

The `binder/edit.sh` script will do the same, but also bind-mount the tutorials into the container so that they can be edited in the Jupyter environment.

When preparing a release, run `binder/exec.sh` and commit the results into the repository.

11.3 How to Release a New HTMap Version

To release a new version of HTMap:

1. Run `binder/exec.sh`, check that they executed correctly by loading them up in a Jupyter session, and commit the resulting executed tutorial notebooks into the repository.
2. Make sure that the version PR actually bumps the version in `setup.cfg`.
3. Merge the version PR into `master` via GitHub.
4. Make a GitHub release from https://github.com/htcondor/htmap/releases, based on *master*. Name it exactly `vX.Y.Z`, and link to the release notes for that version (like https://htmap.readthedocs.io/en/latest/versions/vX_Y_Z.html) in the description (the page will not actually exist yet).
5. Delete anything in the `dist/` directory in your copy of the repository.
6. On your machine, make sure `master` is up-to-date, then run `python3 setup.py sdist bdist_wheel` to create the source distribution and the wheel.
7. Install Twine: `pip install twine`.
8. Upload to PyPI: `python3 -m twine upload dist/*`. You will be prompted for your PyPI login.

HTMap's default Docker image is defined by the `docker/` directory in this repository. It is built automatically by Docker Hub, see the builds page. The Binder-served tutorials also use an image built by Docker Hub: see here, and are defined by the `binder/` directory in this repository.

PYTHON MODULE INDEX

h

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W